BERT Base Uncased Inference Workload Characterization and Microarchitectural Exploration on Intel Skylake Processor

A Dissertation as a Partial Fulfillment for

Master of Science in Mathematics

Vishwanath Saikiran Shetiya

Regd. No: 23017

SRI SATHYA SAI INSTITUTE OF HIGHER LEARNING

(Deemed to be University)

Department of Mathematics and Computer Science

Prasanthi Nilayam Campus - 515134

April - 2025

SRI SATHYA SAI INSTITUTE OF HIGHER LEARNING
(Deemed to be University)

Department of Mathematics and Computer Science
Prasanthi Nilayam Campus

CERTIFICATE

This is to certify that this Dissertation titled **BERT Base Uncased Inference Workload Characterization and Microarchitectural Exploration on Intel Skylake Processor** submitted by Vishwanath Saikiran Shetiya, 23017, Department of Mathematics and Computer Science, Prasanthi Nilayam Campus is a bonafide record of the original work done under my supervision as a partial fulfillment for the Degree of Master of Science in Mathematics.

[illegible]
....
Dr. R. Raghunatha Sarma
Dissertation Supervisor

Countersigned by

[illegible]

Place: Puttaparthi

Dr. (Ms.) Y Lakshmi Naidu
Head of the Department

Date: April 10, 2025

DECLARATION

The Project titled **BERT Base Uncased Inference Workload Characterization and Microarchitectural Exploration on Intel Skylake Processor** was carried out by me under the supervision of Dr. R. Raghunatha Sarma, Department of Mathematics and Computer Science, Prasanthi Nilayam Campus as a partial fulfillment for the Degree of Masters of Science in Mathematics and has not formed the basis for the award of any degree, diploma or any other such title by this or any other University.

Vshetiya

……………………………..

Place : Puttaparthi

Vishwanath Saikiran Shetiya

Regd. No: 23017

Date : April 10, 2025

II M.Sc Mathematics

Prasanthi Nilayam Campus

Acknowledgments

I am grateful for working on this project for the past few months. It has helped me acquire knowledge and information about the field I am passionate about.

First and foremost, I express my gratitude to Bhagawan Sri Sathya Sai Baba, my constant source of inspiration. His teachings and lessons have helped me to pursue this project properly.

I express my gratitude to my guide, Dr. R. Raghunatha Sarma, who has constantly supported this project and ignited my passion for further exploring the field.

To Mr. Dibyam Pradhan (Principal CPU Architect, ARM), Mr. M. Naveen (Senior Member of Technical Staff, AMD), Mr. Mangala Prasad (Hardware Developer, IBM), and Mr. Sai Aravind S. V (Graduate Engineer, ARM) I extend my heartfelt gratitude for their invaluable support throughout my journey. Their guidance and insights have been instrumental in shaping me and this work.

The contributions of the authors of the referenced article have significantly influenced the direction and depth of this work. Their insights and research have provided a strong foundation upon which this study is built. I sincerely acknowledge and appreciate their valuable efforts and contributions.

I sincerely thank Dr. N. Uday Kiran and all professors of the Department of Mathematics and Computer Science, SSSIHL, for their guidance, support, and resources that greatly contributed to this work.

Finally, I sincerely thank my parents, friends, and well-wishers for their unwavering support and love throughout my life.

Abstract

Deep learning models like transformers have significantly improved performance on tasks such as language understanding, sentiment analysis, question answering, machine translation, and text generation. While hardware like CPUs and GPUs accelerate deep learning tasks, CPUs can outperform others for some of the deep neural network inferences under certain conditions. This makes them favorable for inferences of certain applications. However, achieving the efficiency on running of the transformer models on CPUs remains a challenge, particularly when we have limited hardware resources and trying to balance with low latency. There's a significant potential to improve the current hardware capabilities of CPUs for efficient running of the transformer models.

In this project, we have carried out experiments to characterize the micro-architecture of the deep learning model on Intel Core 8th-generation CPU platform. We study how the transformer architecture works. Hotspot analysis was conducted on these workloads to know the top hotspot function of the workload. A top-down performance analysis approach was used to explore performance at the micro-architecture level.

This research gives insightful information and identifies key areas for optimizing and improving microarchitectural efficiency for better performance of the applications. This project examines various experiments conducted, questions explored, and insights derived from them.

Contents

List of Figures

Chapter 1

Introduction

Due to the rise of Deep learning, there has been significant progress in many domains, including natural language processing, computer vision, Reinforcement Learning, autonomous systems, etc. The Transformer models represent a significant advancement in deep learning, producing a powerful and flexible architecture for various Natural Language Processing tasks. Now, it is even being extended to computer vision. However, the capabilities that empower these models also present a challenge - their computational complexity often hinders their deployment on resource-constrained devices.

Transformer models, while powerful, are computationally demanding due to their complex architecture and attention mechanism. The attention mechanism is central to their operation and involves calculating the similarity between every pair of elements in the input sequence. This can be computationally expensive, making it challenging for devices with limited resources, like smartphones and systems. Running these models on such devices often leads to high power consumption, short battery life, and delays in processing.

Among the Transformer models, we have the Bidirectional Encoder Representations from Transformers (BERT) base uncased model, which is an encoder-based transformer model that has achieved state-of-the-art results on a wide range of Natural language processing tasks like question answering, text classification, and named entity recognition. However, the BERT base uncased is a transformer model that is also computationally demanding and can limit its deployment on resource-constrained devices.

To address these challenges, Micro-architecture optimizations are crucial for efficient inference of deep learning models. Micro-architecture is like the blueprint of the processor's internal structure. Hence, optimizing it will be like making changes in its blueprint to improve the processor's efficiency for efficient inference of the BERT Model. In this way, a deeper understanding of how the model interacts with the underlying hardware is gained. This understanding allows us to identify potential bottlenecks within the processor that hinder efficient execution and open the path of opportunities for improvement.

We use various techniques to characterize the micro-architecture behavior of the BERT base uncased model. Profiling tools like Intel's VTune profiler can be used to analyze the model's execution on a particular processor and find those areas that consume a significant amount of processing cycles and those areas creating the bottleneck to the performance and efficiency of the model.

Micro-architectural analysis helps us bridge the gap between hardware and software by determining how the software is running on the processor. Thus, the performance can not be improved only by making the software efficient but also by making some micro-architectural changes. Since the CPU is the Central component of all computing resources, the project focuses on performance analysis for CPUs. With the Intel processor, the Workload analysis is carried out using Intel's VTune Profiler.

1.1 Problem Statement

The objective of this project is to

1. Micro-architectural characterization of BERT base uncased model and exploring its micro-architectural improvement opportunities and

2. Identify bottlenecks at the micro-architectural level and suggest improvement opportunities.

Chapter 2

Background

2.1 Deep Learning: Concepts, Architectures, and Applications

2.1.1 Introduction

Deep learning uses multilayered neural networks, called deep neural networks, to simulate complex decision-making like the human brain [2]. The main difference between deep learning and machine learning is that machine learning uses one or two layers, whereas deep learning uses three or more layers.

In deep neural networks, we have multiple layers, among which we will be doing a forward pass that is similar to mathematical functions. The input layer is where the model takes input for processing. Like how the data changes when an input is given to the mathematical function, similarly, as the data passes by different layers during the forward pass, the data changes, helping the model to learn what it is. The output layer is where the final prediction or the classification would happen. Then, we do backpropagation to improve the predictions of the model and reduce the loss that is calculated after each iteration. Where we update weights and biases the parameters of the model so that it learns to predict.

Multiple frameworks like PyTorch and TensorFlow would help us to make deep learning systems.

Modern artificial intelligence applications use deep learning models in various fields, such as image classification, image generation, natural language processing, etc. These above tasks use deep learning architectures like Convolutional neural networks (CNNs)[3], Recurrent neural networks (RNNs)[4], Transformer models[5], and Diffusion models.

Among the different deep learning models, Large language models are those systems trained on a massive amount of text data. This is done so that they develop a human-like understanding of languages. Different architectures would process the same data in different ways. Like, RNN would process it sequentially, LSTM's also do sequentially but have larger information than RNN by using gates. Transformers use self-attention mechanisms to process data in parallel.

2.1.2 Deep Learning Workflow

The deep learning workflow is made up of getting ready with data, building a model, model fitting, model prediction, evaluating the model, and improving through experimentation, inference, and deployment.

As discussed before, we have many frameworks with a similar workflow to build code for deep learning models. Frameworks like PyTorch, Keras, and TensorFlow make it easy to build a deep learning model following the workflow.

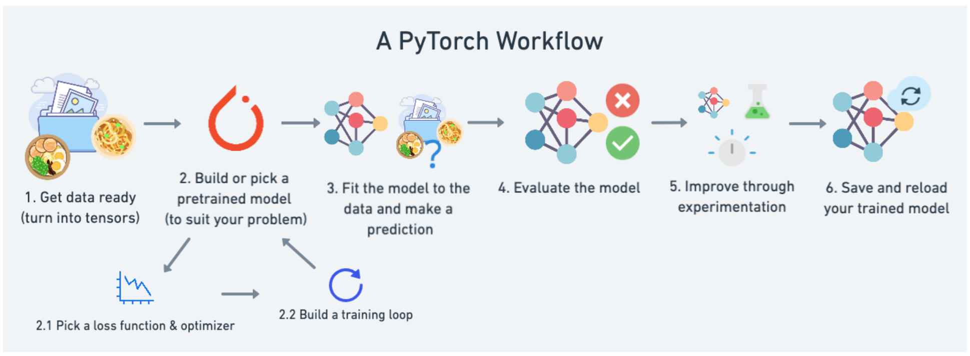

A PyTorch Workflow
1. Get data ready (turn into tensors)
2. Build or pick a pretrained model (to suit your problem)
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentation
6. Save and reload your trained model
2.1 Pick a loss function & optimizer
2.2 Build a training loop

Figure 2.1: Pytorch Workflow

For our project, we have chosen the PyTorch framework. PyTorch helps take care of GPU acceleration that is making the code run faster.

PyTorch helps to represent different kinds of data like image, text, and audio in the form of tensor - a form of data that a deep learning model can understand.

The figure 2.1 illustrates the PyTorch framework workflow[6]

1. Getting Data Ready: Our first step would be to get ready with the data. Processing according to our requirements.

2. Building a Model: We will create a model according to our needs. The model should have layers such that it would be able to learn the patterns in the data that we input. We should choose a relevant loss function and optimizer that would facilitate the training of our model.

3. Training the Model: This is the stage where the model learns the pattern. Optimizes and gets used to the patterns of the data using the backpropagation algorithm.

4. Evaluation of the Model: At this stage, we evaluate how well the model is performing. We use data of the same kind, but we will show different data, which in machine learning terminology is called test data.

5. Saving and loading the model: We should be able to deploy the model anywhere and use it. Hence we will be saving it so that we can load it anywhere else.

2.2 Natural Language Processing

Natural Language Processing (NLP) is a branch of artificial intelligence that would help in making models that would understand language in a way humans do. The models would process the human language[7].

Nowadays, these large language models(LLMs) have grown to have the ability to generate images. They are used by many search engines for facilitating improved search results, chat bots for customer services, voice-operated GPS systems, etc.

2.2.1 What is Natural Language Processing (NLP) Used For?

Sentiment Analysis: Models for such analysis are used to classify people depending on their emotions and feelings[7].

Machine Translation: Models here are trained to translate a given sentence of a language to another language. Google Translate is one prominent example.

Text Generation: Also known as natural language generation (NLG), produces text based on previous texts generated. Features like auto-complete are because of this.

Named Entity Recognition: Such models aim to recognize some of the entities, such as names, places, locations, etc. For the given text, the model would help us to find words that are names, words that are places, etc.

Question answering: Models that are trained to answer questions asked by humans.

Some of the large language models are GPT, BERT, PEGASUS, LLAMA, LaMDA, etc. Most of the large language models use transformer architecture, which is based on a self-attention mechanism.

2.3 Transformers - Attention is All You Need

2.3.1 Introduction

The Transformer is a neural network architecture based on a multi-head attention mechanism. It was introduced in 2017 [5] and aims to solve sequence-to-sequence tasks while easily handling long-range dependencies. Transformers were generally used for various natural language processing (NLP) tasks and are now used for image classification, such as vision transformers.

2.3.2 What distinguishes the Transformer architecture from other LLM Architectures?

Transformers are a type of neural network architecture used in natural language processing (NLP). They have achieved state-of-the-art performance on various tasks such as machine translation, text summarization, and question answering.

What sets transformers apart from other language model architectures is their ability to capture long-range dependencies in sequential data and their attention mechanism.

Key Features of Transformer Architecture

1. **Self-Attention Mechanism:** The recurrent neural network (RNN) and long-range short-term memory (LSTM) networks do sequential processing, whereas the Transformer architecture uses its self-attention mechanism and makes sure that each token in the input interacts with every other token of the input.

2. **Parallel Processing:** Processing of input sequentially might cause slower training of the model. The transformer can process entire sequences in parallel, making them more efficient than RNNs and LSTMs, which process them sequentially.

3. **Vanishing Gradients:** In RNNs, derivatives going smaller leads to vanishing gradient, leading to slower training of the model. To improve this they came up with LSTMs that used memory cells and gates to overcome the problem of vanishing gradient, but it still lacked parallel processing. Hence, due to the attention mechanism and residual connection, the Transformer architecture overcame all these problems.

2.3.3 Transformer Architecture and How Transformer Works?

Transformer Architecture

The Transformer Architecture[5] has an encoder and a decoder block. The encoder takes in our input and outputs a matrix representation of that input. This encoded matrix representation of the input will have contextual information regarding the input that we have given. The decoder takes in that encoded representation and iteratively generates an output.

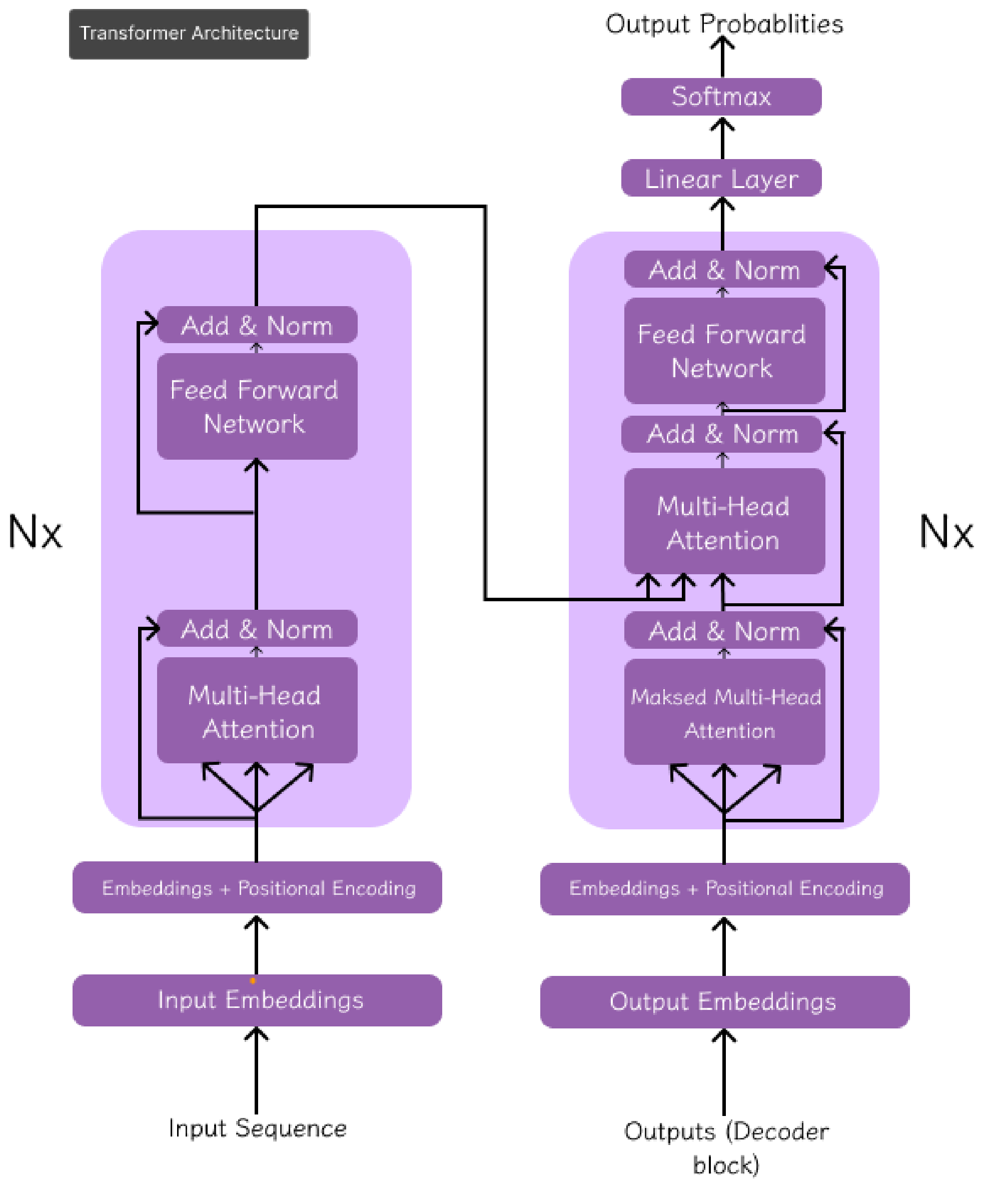


Figure 2.2: Transformer Architecture

However, the encoder and the decoder are stacked with multiple layers that have the same number of blocks for both the encoder and decoder stacks. All encoders present the same structure, and the input gets into each of them and is passed to the next one. All decoders also present the same structure and get the input from the last encoder and the previous decoder.

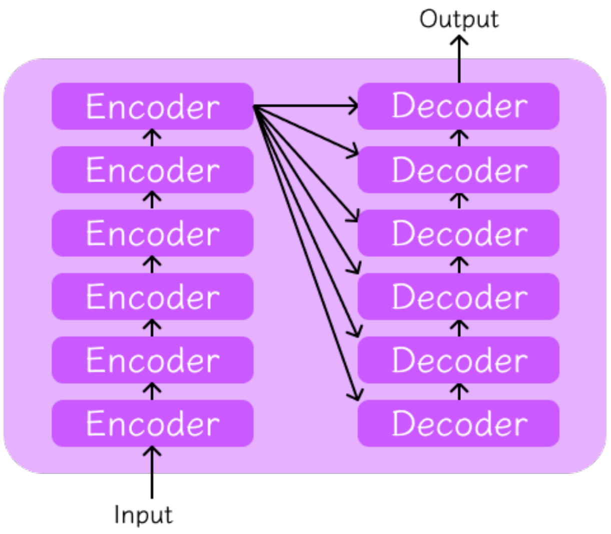


Figure 2.3: Encoder and Decoder Stack with N=6

2.3.4 Transformer Workflow

The Encoder Workflow

The primary function of the encoder is to transform the input tokens into contextualized representations. The encoder begins by converting the input tokens into vectors using embedding layers. These embeddings capture the semantic meaning of the token.

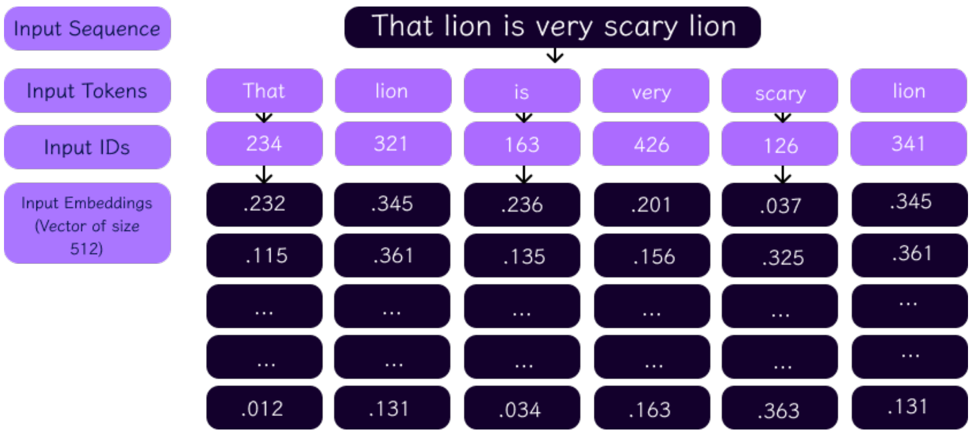


Figure 2.4: Embedding Layer

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2.1}$$

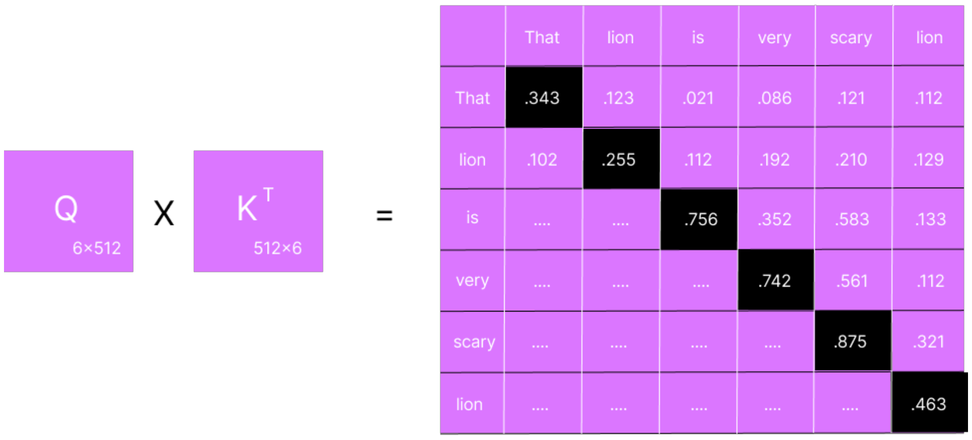
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2.2}$$

For the embeddings we have above, we add the positional encoding to get the relative position of each of these tokens.

Then, this input X was projected into query (Q), Key (K), and value (V) matrices [5]:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{2.3}$$

The encoder block has two main layers: one is a multi-head self-attention, and the other one is a feed-forward neural network. In multi-head self-attention first, we do the scaled dot product of the query and key vectors of the input sequence.



	That	lion	is	very	scary	lion
That	.343	.123	.021	.086	.121	.112
lion	.102	.255	.112	.192	.210	.129
is	....	....	.756	.352	.583	.133
very	....	....	....	.742	.561	.112
scary	....	....	....	....	.875	.321
lion	....	....	....	....	....	.463

Figure 2.5: Dot Product of Query and Key Matrices

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Then, we do the softmax to get the values between 0 and 1. As we get the attention scores, we multiply them with the values matrix to get the output that speaks about the relationship between every token of the input sequence.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The same calculations happen in every head in that layer. The process is illustrated in figure - 2.6.

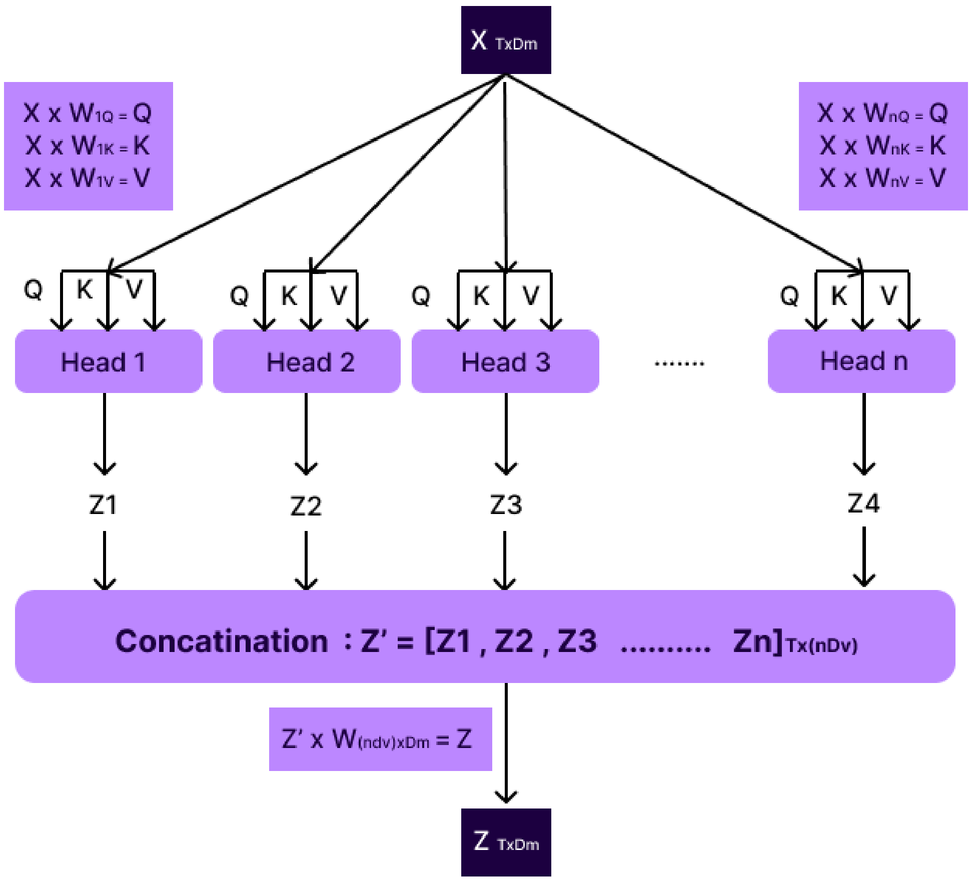


Figure 2.6: Multihead Self Attention Mechanism

As we see above, we will have the dk and dv dimensions of the weight/projection matrices equal to dm/h, where h is the number of heads. Then, from each head, we get Zis, which are then sent to the sub-layer where they are concatenated. Finally, using one more weight matrix, we project the output of the concatenation sub-layer to the input dimension.

Diffrent kinds multi-head attention:[5]

1. **Encoder-Decoder Attention:** Queries from the decoder attend to all positions in the encoder output. So that the decoder gets the language understanding from the output of the encoder for decoding.

2. **Self-Attention in Encoder:** In this Attention mechanism, each token in the encoder attends to all other tokens in the input sequence.

3. **Self-Attention in Decoder:** Each token in the decoder attends to all the previous tokens in the input. This would showcase the auto-regressive property of the models that would use the decoder block of the transformer. This is done by a masking mechanism that masks the future tokens.

Advantages of the multi-head attention[8]

This way due to multiple projection matrices projecting the input to Q, K, and V, we can focus on different aspects of input that would help us to understand various aspects and relevant information between tokens. The model would also adapt to various datasets.

The next sub-layer is a feed-forward network that has 2 linear layers and one activation function, namely RELU.

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

Each sub-layer in an encoder layer is followed by a normalization step. Also, each sub-layer output is added to its input (residual connection) to help mitigate the vanishing gradient problem. Then we do layer norm.

$$x' = \text{LayerNorm}(x + \text{MultiHead}(Q, K, V)) \tag{2.4}$$

$$y = \text{LayerNorm}(x' + \text{FFN}(x')) \tag{2.5}$$

The output of the final encoder layer is a set of vectors, each representing the input sequence with a rich contextual understanding. This output is then used as the input for the decoder in a Transformer model.

The Decoder Workflow

We here send them the output of the encoder block and the output of the previous decoder block. Similar to the encoder block it obtains the positional embeddings, and positional encodings are found and added to find the relative positions of the word, to know the order of the sequence.

The input first goes through marked multi-head self-attention. Where each token will be able to attend to those tokens that are before it. We then multiply it by vector V and stack up all heads. Then, we have a second multi-head attention, which takes the input as the query vectors from the previous sublayer and value and keys from the output of the encoder block.

Each sub-layer in an encoder layer is followed by a normalization step. Also, each sub-layer output is added to its input (residual connection) to help mitigate the vanishing gradient problem. Then we do layer norm.

Passed on through linear layer to project on to vocabulary and then softmax to get the probability values and get the highest likelihood values. This can be repeated or stacked up N times, where each time it gets input from the output of the previous block.

Summary

This subsection on transformers speaks about the flow and architecture of transformers. We also learned that transformer architecture is faster and more efficient than traditional RNN architecture.

2.4 Chapter Summary

In this chapter, we covered the significance of deep learning models and how they are different from machine learning models. We saw different kinds of architectures that would help in different tasks. We saw the different stages of coming up with a deep learning model.

Then, we had the overview of Natural Language Processing (NLP). We saw how the transformer uses its self-attention mechanism to work with data in parallel. The work done in this field by academic and industrial researchers will be discussed in the next chapter.

Chapter 3

Literature Survey

In this chapter, we explore notable advancements by researchers in deep learning and computer architecture, with a focus on transformer-based models such as BERT (Bidirectional Encoder Representations from Transformers). Then, this section will conclude with our finalized problem statement.

3.1 BERT: Bidirectional Encoder Representations from Transformers

3.1.1 Introduction

Bidirectional Encoder Representations from Transformers are based on transformers, a deep learning model that takes a sequence of input and gives a sequence of output. As the name suggests, the BERT Model is encoder-based. It was introduced by Google in 2018 [9] and helped them in the search mechanism very well. The BERT model since it is an encoder-based model, is generally known for understanding the input sequence and context. We have different types the two main ones are BERT Base (Cased and Uncased) and BERT Large (Cased and Uncased). Others are the optimized or fine-tuned versions of these, e.g., Robustly Optimized BERT, SpanBERT, etc. In this Project, we will be dealing with the BERT base uncased model.

3.1.2 BERT Base Uncased Architecture

It is an encoder-based model. The Bert Base uncased model is made up of 12 blocks of transformer encoder blocks[9].

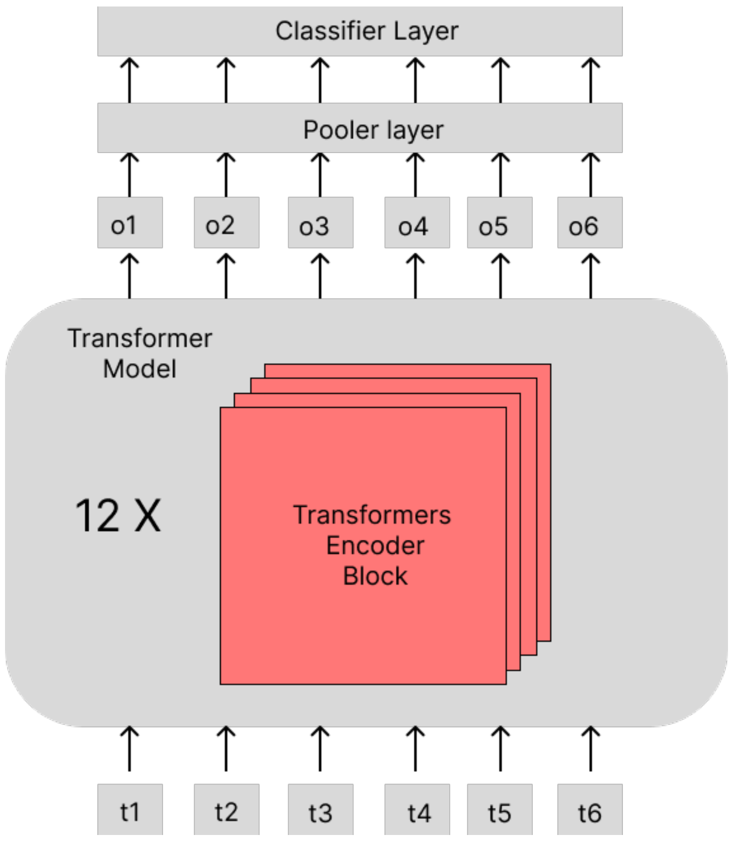


Figure 3.1: BERT Base Uncased Architecture

Each encoder block consists of Multi-head Self-attention with 12 Heads each which helps in capturing different aspects of the input. Feed Forward network with GELU activation function. The hidden size is 768. There are a total of 110M Parameters. The "bidirectional" aspect of BERT refers to how it considers both left and right tokens when processing a token. A pooler layer follows the Transformer Model stack, which gives out the pooled output of a token that acts as an aggregate representation of the whole sequence. Depending on the task or fine-tuning, a classifier layer is introduced.

3.1.3 Input and Output representation

Input Representation

BERT is trained on datasets from Wikipedia and book corpus, and it has a vocabulary list of 30,000 tokens. For a given sequence, we have tokens based on WordPiece tokenization. These tokens are mapped to an index in BERT's vocabulary, where those indices are assigned to a vector in the embedding space based on the WordPiece tokenization. A token called [CLS] is added at the beginning of each input sequence. [CLS] is a classification token used specifically for classification tasks (used in pertaining). The output of this token will be an aggregate representation of the entire input sequence[SEP] is also used in the input sequence to separate the segments in input sequences. It is used even at the end of the input sequence (i.e., a single segment).

Output Representation

Since we have the pre-trained model, Hidden states are the output from the transformer block, where we get the output of each token. The hidden state of [CLS] holds the contextual information of the input sequence. After going through the pooler layer, the pooled output is the vector representation of the [CLS] token that acts as the aggregate representation of the input sequence.

3.1.4 Bert Base Uncased Pre-Training Phases

The pertaining is done in two ways, using masked language models and next-sentence prediction. This is a self-supervised way of learning. These help in the general understanding of the language the model is trained on.

Masked Language Model

The model learns to predict the missing word or masked word based on the information or the context of surrounding words. When given input to the model, 15% of the input is chosen and they are replaced 80% of the time with [MASK], 10% of the time with different tokens, and 10% of the time with the same token. [CLS] is added at the beginning and [SEP] is added at the end. The model masks according to the condition given above.

A classification layer is added to the model that projects the output of the model over the vocabulary. Each vector goes through a linear layer of the classification layer and gets projected over the vocabulary. Masked token outputs go through a softmax layer during MLM pertaining, and we get the probability distribution. Generally, the one with the maximum probability sits in the masked position.

Since it is the training phase, Loss is also calculated using ground truth and predicted probability distribution using cross-entropy. The model is then optimized using AdamW Optimizer to predict the correct token.

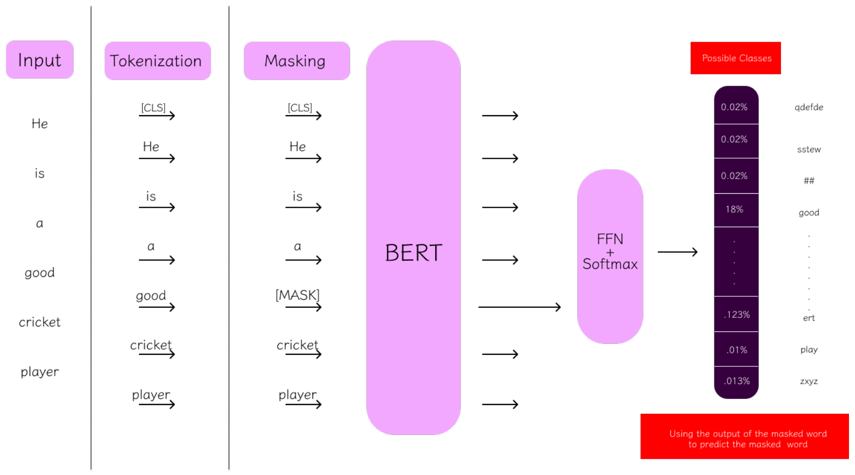


Figure 3.2: BERT: Masked Language Model

Next Sentence Prediction

The model learns to predict whether or not two sentences appear consecutively. [CLS] is added at the beginning and [SEP] is added in between and at the end. The sentence that follows sentence A is 50% of the time the next sentence and 50% of the time some random sentence. The output of the [CLS] token that acts as the aggregate representation of the input is projected to a vector of 1×2, giving us the logits, then passed to softmax to give a probability distribution.

Since it is the training phase, Loss is also calculated using ground truth and predicted probability distribution using cross-entropy. The model is then optimized using AdamW Optimizer to predict the correct token. In this manner, the BERT learns sentence-level understanding.

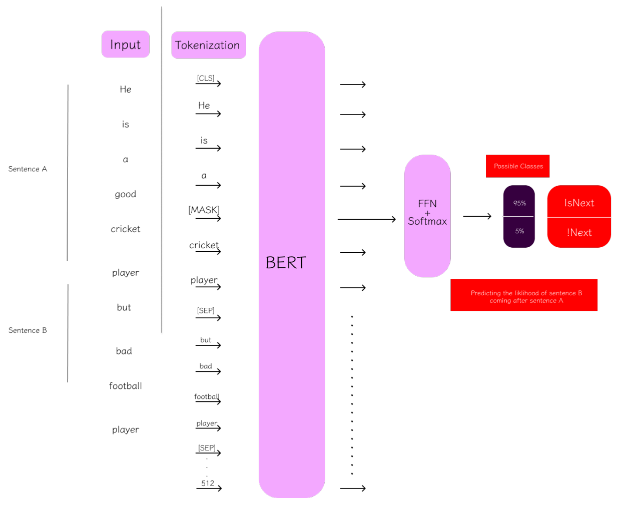


Figure 3.3: BERT: Next Sentence Prediction

Conclusion: Pre-training phase

BERT optimizes these two objectives simultaneously, they are generally optimized by providing a large corpus of text so that the model learns the contextual relationship between the words and sequences. This helps in downstream tasks as it has the word level and sentence level knowledge.

Fine-tuning

It is the second phase of training - fine-tuning. Here, the models are trained for specific tasks. For fine-tuning, since it is generally supervised, we need task-specific labeled data. Since we are making the model suited to a specific task we might have to add some layers specific to the task. Some examples of tasks for which the model is fine-tuned are Sentiment analysis, text classification, and named entity recognition.

3.1.5 Summary

In this section, we studied how the BERT model works, how it takes input, what its pre-training phases are, and how the BERT model can be fine-tuned for tasks such as text classification, translation, disambiguation of words with different meanings, sentimental analysis, paraphrase Identification, and many more.

3.2 From Large Language Models to Machine

This section provides an architectural and performance-centric analysis of how modern large language models (LLMs)- like BERT perform on CPU hardware, particularly those models with SIMD capabilities. Researchers are examining and trying to identify bottlenecks either at the software or hardware level and optimize inference efficiency.

This study of such research focuses on several key aspects of LLM inference on CPUs:

1. **Performance Metrices:** When LLMs execute on CPUs, they give out some of the important performance characteristics on certain metrics. Key metrics include the number of instructions executed, branch predictions, and cache misses. Some of these metrics influence the performance of the model.

2. **Layer-Wise Time Analysis:** This analysis helps us to know how much time each layer of the model is taking. This analysis could help us in identifying the bottlenecks. In this manner it would direct the researchers to focus on that layer to look for some bottlenecks.

3. **Dynamic Sparsity:** LLMs exhibit dynamic sparsity during inference. This keeps the neurons active, depending on the input. Hence, the hardware could skip some redundant computations, presenting an optimization opportunity for better CPU-based inference.

4. **Data Movement:** Transformers work differently with the data. Things like multihead attention involve structured computations that impact data flow and memory access patterns. We can look for these patterns and organize the way data flows so that the inference can be efficient.

5. **Hardware-Specific Optimization:** The optimization will not be only at the software level. It can also be at the hardware level, like removing structural hazards by trying out with extra execution unit, better cache utilization. This means that we will have to align the hardware to the model's

computational characteristics.
The goal of this work is to offer recommendations for enhancing deep neural networks' efficiency at the hardware and software levels. Here, we are using the SIMD CPUs for our analysis. The results may be used to locate deep neural network's performance bottlenecks and may even inspire the creation of new hardware architectures for efficient running of deep neural networks like Transformers.

GPUs are extremely powerful for training neural networks because they can process large amounts of data in parallel. This makes them ideal for the training phase, which involves huge datasets and complex calculations. However, GPUs are not always the best choice for the inference phase—when the model is used to make predictions. Inference usually deals with smaller amounts of data and needs to give fast, real-time responses. In these situations, CPUs can sometimes be more efficient than GPUs because:

1. In training, we have many tasks that have to be executed in parallel. However, inference tasks are often sequential, meaning they have to be executed one after the other. This will limit the GPU's parallel processing capabilities.

2. CPUs are better at handling small, quick tasks.

3. CPUs often have lower power consumption.

4. CPUs are more readily available in everyday devices like laptops, smartphones, and servers.

Summary

In this section, we saw what all should be considered when the large language models interact with hardware. We also saw that, given a chance, we should be able to find the bottleneck at the software or at hardware level and try to find optimization opportunities for efficient running of the model.

3.3 Choices Regarding the Project

3.3.1 Deep Learning Phases: A Comparison of Training and Inference

As we saw above, in deep learning, workflow has two main key phases: training and inference.

Training

Training is the process where the model learns from a large dataset. It's like a student studying for an exam. In this phase, the model tries to analyze the data and, adjust the internal parameters to reduce the loss function, leading to finding the patterns that will help them to perform well in the future. Training requires a lot of computational power because the model is constantly adjusting its internal weights and learning from the vast amount of data. Hence, for such purposes, hardware like GPUs are designed to handle large amounts of data in parallel.

For example, we have models like BERT or GPT, which process billions of words, adjusting billions of parameters to make accurate predictions.

Inference

Once the model has been trained, it is used for inference to evaluate how it works so that we can deploy it for applications. Here, in this phase, the model predicts the unseen data. This stage is like a student taking a test. Model usese the knowledge gained during the remaining phase and uses those learned parameters to predict on this data.

Relative to the training phase, it needs less computation power because the model has already learned, and we need not adjust the parameter anymore. We also send the input sequentially for prediction, so lesser data compared to the training phase. Hence, CPU is often sufficient for inference tasks.

3.3.2 Why BERT Base Uncased Model

From the section 2.3, we saw how the transformer works, and we saw in that section how it handles data in parallel and that the architecture is better than other previous models like RNNs and LSTMs by using its self-attention mechanism. BERT Base uncased Model is one of the of those Transformer models.

The BERT Base Uncased Model, as the name suggests, the Base here means it is a small BERT, which means it is faster for training and inference. Compared to larger BERT models (e.g., BERT Large), BERT Base Uncased is less computationally expensive, making it suitable for environments with limited resources. It acts as a strong baseline to start with for various NLP tasks. The "Uncased" version means it is pre-trained on a dataset where all words have been converted to lowercase. This can be beneficial if your task is not sensitive to the case. Since it is trained on a large data set, we can fine-tune it using a small data set, which is helpful for transfer learning.

Summary

In this section, we introduced the two primary phases of deep learning: training and inference. We discussed how training involves learning from large datasets and requires high computational power, typically supported by GPUs, whereas inference focuses on fast and efficient predictions using trained models, often on CPUs.We also presented the rationale for choosing the BERT Base Uncased model for this project.

The problem statement for this project is the result of our research, literature survey, and informed decisions. In this work, we aim to explore and understand the performance characteristics of the BERT-based uncased model during inference on CPUs. To achieve this, we will conduct a detailed performance analysis using Intel's VTune Profiler to characterize how the model utilizes the processor. Furthermore, in the subsequent chapter, we will outline the performance analysis methodologies employed and present a top-down approach to investigate CPU microarchitectural behavior during BERT inference.

Chapter 4

Methodology and Profilers

In this chapter, we will discuss the methodology and profiling tools selected to facilitate our experiments.

4.1 Top-Down Methodology

The top-down methodology is introduced in a paper titled "A Top-Down Method for Performance Analysis and Counters Architecture" [1] by Ahmad Yasin.

4.1.1 Objective

To increase the performance, nowadays, the CPUs use a variety of techniques like hardware prefetching, super-scalar, out-of-order execution, and speculative execution of instruction. PMUs - performance monitoring units, an on chip subsystem would help us to get the hundreds of performance events of a chip. Among those, it is difficult to identify the performance bottleneck.

$$\text{Stall\_Cycles} = \sum_i \text{Penalty}_i \times \text{MissEvent}_i \tag{4.1}$$

According to the traditional methods, to estimate the CPU stalls we simply multiply the delay that is penalty latency by the number of miss events. But this will not work well on modern out-of-order processors. This is because the processors can execute instructions at the same time, which causes overlaps in delays. Apart from this, the cashes also have different hit rates. Hence, the old techniques will not be able to calculate the stalled cycles accurately. Hence, new methods like the top-down approach are needed to pinpoint performance bottlenecks in modern CPUs.

4.1.2 The Hierarchy

The Top-Down Microarchitecture Analysis Method (TMA) is a novel approach to identifying performance bottlenecks in the modern out-of-order processor cores. This approach employs a hierarchical structure of performance counters to systematically detect and classify inefficiencies and efficiencies within the CPU pipeline[1].

The pipeline of a modern high-performance CPU is quite complex. The pipeline is mainly divided into two halves, the Front-end and the Back-end. The front end is responsible for fetching the instructions and decoding them into one or more low-level hardware operations called micro-ops (uOps). These uOps are then fed to the Back-end of the processor. After the allocation of the uOps to the Back-end, Back-end is responsible for monitoring when the uOp's data operands are available and executing the uOp in an available execution unit. The completion of uOp's execution is called retirement, and it is the stage where the results of the uOps are committed to the architectural state - CPU registers or written back to memory. Usually, most of the uOps pass completely through the pipeline and retire, but sometimes, speculatively (execution of instructions based on branch prediction) fetched uOps may get canceled before retirement.

The partition of the pipeline we saw above acts as a baseline for the Top-Down method categories. During any cycle, a pipeline slot can either be empty or filled with a uOp. If a slot is empty during one clock cycle, this is called a stall. TMA finds the high-level causes of all program execution stalls for each hotspot.

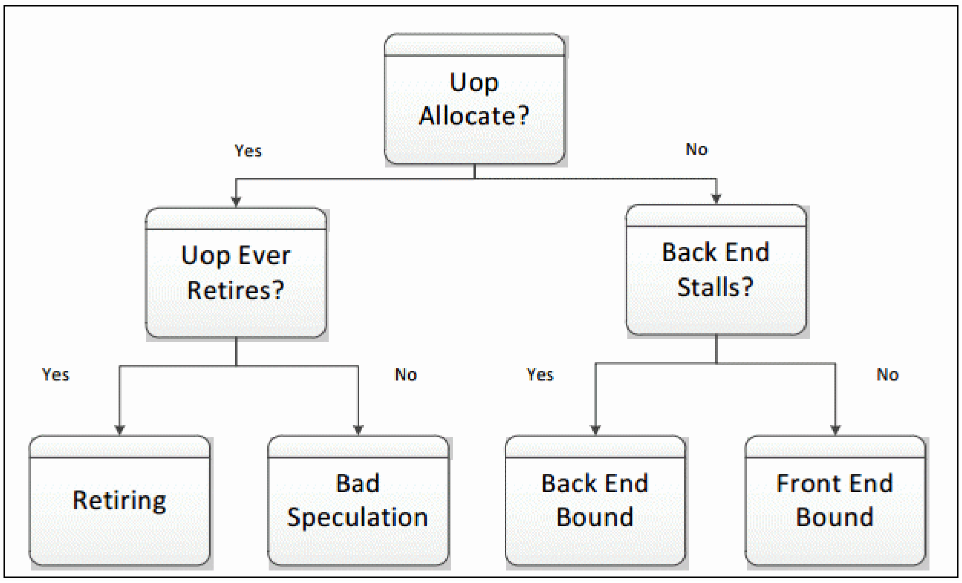


Figure 4.1: Top-Down Micro-architecture analysis [1]

One of the four elements Front-End Bound, Back-End Bound, Retiring or Bad Speculation [1] may be the cause of the bottleneck. How this happens from the perspective of uOps is discussed in the figure. 4.1.

If the processor is not stalled, then a pipeline slot will be filled with a uOp at the allocation point. If the uOp is allocated, this means that it has crossed all the hurdles of Back-End and Front-End bounds. This means that if it is allocated, it has to retire or be discarded if it was a bad speculation. But the challenge comes when the uOps is not allocated. Either it will be because of Front-End or Back-End bound. This means either the backend was overloaded and was unable to allocate resources for fresh uOps, or we were unable to fetch and decode the instruction.

A high Retiring value means the CPU is doing a lot of productive work, but even in such cases, performance may still be suboptimal due to other hidden issues. Take for a non vectorized code it will show a high retiring value but we might be missing out on some of the performance gains like the code may not be optimal. In such cases, the absence of vectorization often implies missed opportunities for performance improvement, particularly on modern CPUs that support SIMD instructions like AVX. In such cases, the high Retiring value can serve as a useful indicator that vectorization could further enhance throughput and efficiency.

Each high-level categories discussed above have deeper categories. Figure.4.2 provides a more detailed assessment of the program's CPU performance constraints.

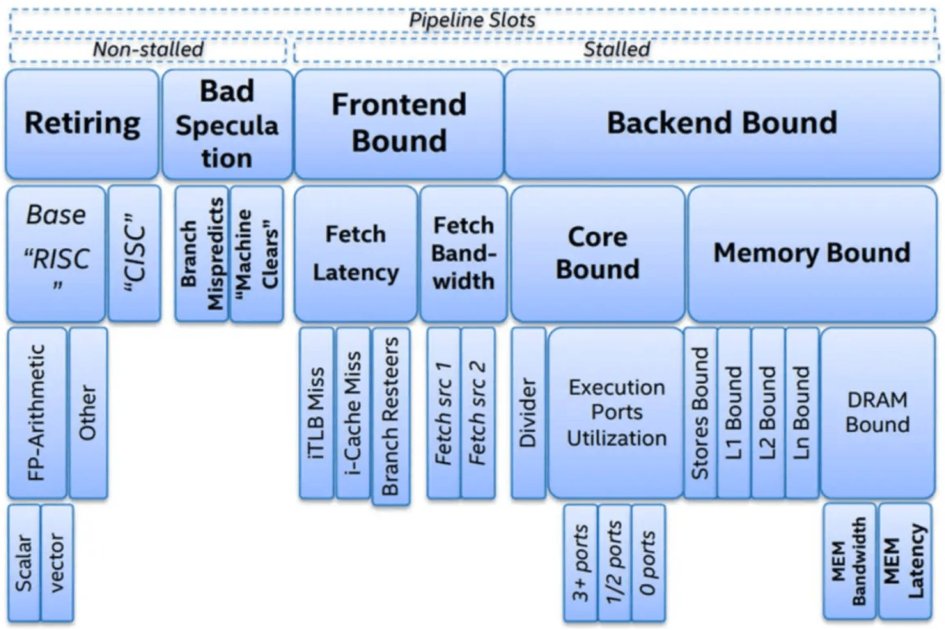


Figure 4.2: Top-Down Micro-architecture analysis [1]

We run workloads multiple times, focusing on different metrics to refine our analysis. For instance, we first gather information for the following four major buckets: Bad speculation, front-end bound, back-end bound, and retiring.

Let us assume that we discovered that a significant portion of program execution was limited due to core execution resource contention, specifically due to poor port utilization (a subcategory of Backend Bound as shown in Fig. 4.2). In such a case, we would repeat the experiment, focusing on port stalls and execution unit pressure.

Once we identify Core Bound as the main culprit, the next step is to locate exactly which instruction sequences compete for port usage. The TMA (Top-down Microarchitectural Analysis) approach provides precise PMC (Performance Monitoring Counter) values for each performance bottleneck. Using this, we can locate the code at both the source level and the assembly level which is causing a bottleneck.

In real-world scenarios, there may be multiple bottlenecks. For example, a program may experience both execution unit pressure (Core Bound) and branch mispredictions (Bad Speculation). In such cases, TMA enables deep dives into multiple buckets simultaneously, quantifying the impact of each.

Tools such as Intel VTune Profiler, Linux perf, and AMD uProf are instrumental in computing these metrics for a given workload. Since the project's experiments are being carried out on Intel processors, we have chosen to use Intel's VTune profiler to perform TMA on a BERT Base Uncased model for my research.Information regarding the Intel VTune profiler are provided in the next section.

4.2 Intel's VTune Profiler

4.2.1 Introduction

Intel VTune is a performance analysis tool designed to help developers identify and optimize performance bottlenecks in applications running on Intel processors. It provides a comprehensive suite of tools for profiling CPU, GPU, and memory usage, making it a valuable asset for optimizing code and improving application performance. Code written in C, C++, C#, Python, FORTRAN, Java, and Assembly can all be profiled by VTune. Some important features of VTune consist of identifying the most time-consuming or hot functions in the workload or applications. Code segments that do not efficiently utilize the available processor time.

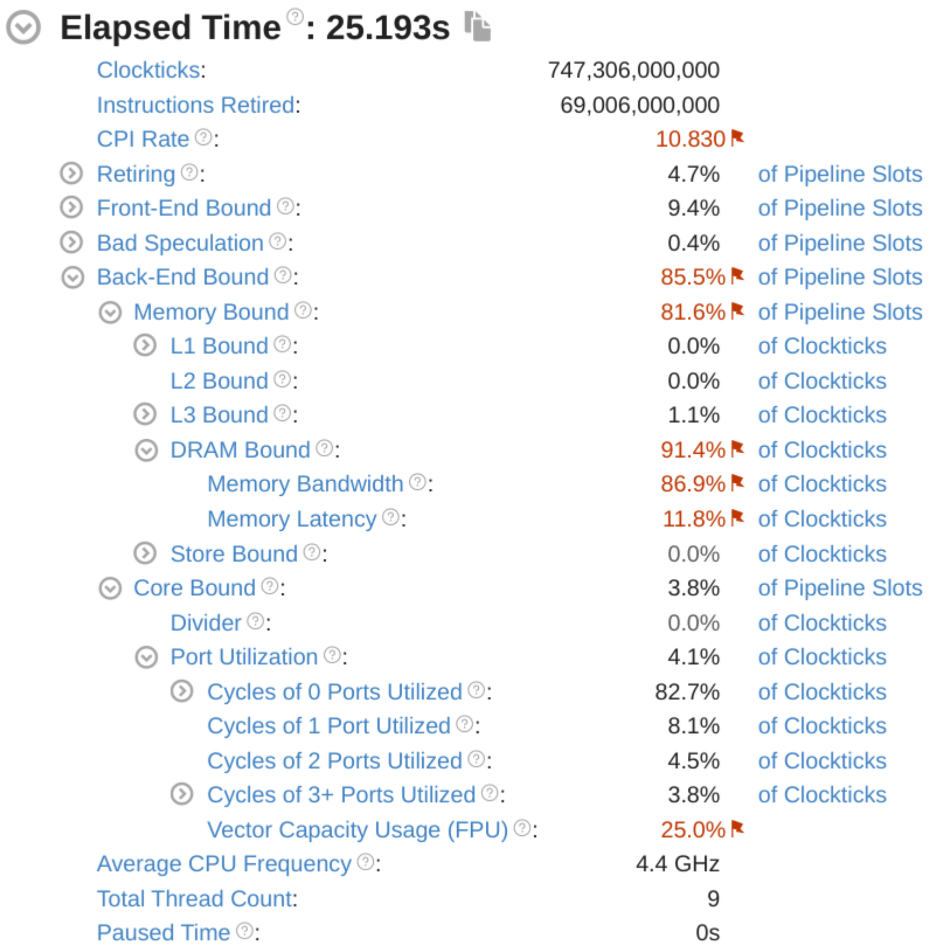
4.2.2 Micro-architecture exploration view

This gives us the recipe to know how an application is utilizing available hardware resources. One way to obtain this knowledge is to make use of performance monitoring units(PMUs) [10].

PMUs are dedicated pieces of logic within a CPU core that count specific hardware events as they occur on the system. Examples of these events may be cache failures or branch mispredictions. These events can be observed and combined to create useful high-level metrics such as Cycles per Instruction.

Modern CPUs employ pipelining as well as techniques like hardware threading, out-of-order execution, and instruction-level parallelism to utilize resources as effectively as possible. Despite this, some types of software patterns and algorithms still result in inefficiencies.

The Top-Down Characterization [1] is a hierarchical organization of event-based metrics in Intel's VTune's Microarchitectural Exploration identifies the dominant performance bottlenecks in an application. It aims to show, on average, how well the CPU's pipeline(s) were being utilized while running an application.



Elapsed Time: 25.193s

Clockticks:	747,306,000,000	
Instructions Retired:	69,006,000,000	
CPI Rate:	10.830	
Retiring:	4.7%	of Pipeline Slots
Front-End Bound:	9.4%	of Pipeline Slots
Bad Speculation:	0.4%	of Pipeline Slots
Back-End Bound:	85.5%	of Pipeline Slots
Memory Bound:	81.6%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	1.1%	of Clockticks
DRAM Bound:	91.4%	of Clockticks
Memory Bandwidth:	86.9%	of Clockticks
Memory Latency:	11.8%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	3.8%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	4.1%	of Clockticks
Cycles of 0 Ports Utilized:	82.7%	of Clockticks
Cycles of 1 Port Utilized:	8.1%	of Clockticks
Cycles of 2 Ports Utilized:	4.5%	of Clockticks
Cycles of 3+ Ports Utilized:	3.8%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Average CPU Frequency:	4.4 GHz	
Total Thread Count:	9	
Paused Time:	0s	

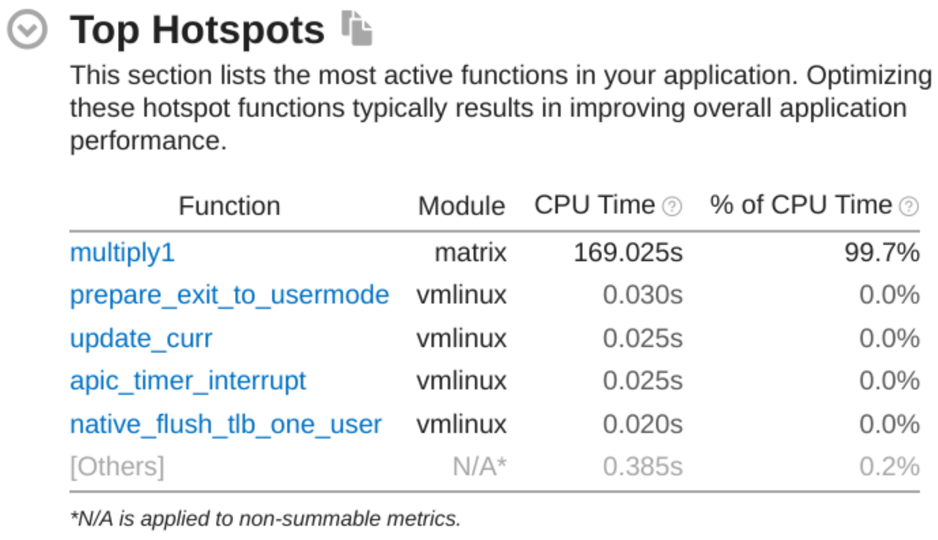
Figure 4.3: Top-Down Micro-architecture analysis for Matrix Multiplication

The Front-end of the pipeline on recent Intel microarchitectures can allocate four uOps per cycle, while the Back-end can retire four uOps per cycle. A pipeline slot represents the hardware resources needed to process one uOp. The Top-Down Characterization assumes that for each CPU core, on each clock cycle, there are four pipeline slots available. It then uses specially designed PMU events to measure how well those pipeline slots were utilized. The status of the pipeline slots is taken at the allocation point, where uOps leave the Front-end for the Back-end.

Each pipeline slot available during an application's runtime will be classified into one of four categories based on the simplified pipeline view of Front-end, Back-end, Bad speculation, and Retiring, like in figure 4.3. Hence, we start from these four categories and go deeper down to find the actual performance bottleneck.

As in figure 4.3 the summary tab shows four main categories like we discussed above and in the section 4.1 analysis. We can click the Summary tab to open the Summary window from the Microarchitecture Exploration viewpoint. The initial segment presents a synopsis of the overall application execution based on hardware-related parameters, expressed in either clock ticks or pipeline slots. Metrics are shown as a Pipe diagram and are arranged in a list figure 4.3 according to the execution categories. We can mouse over the help icon to see a description of a metric

4.2.3 Top Hotspot Functions



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
multiply1	matrix	169.025s	99.7%
prepare_exit_to_usermode	vmlinux	0.030s	0.0%
update_curr	vmlinux	0.025s	0.0%
apic_timer_interrupt	vmlinux	0.025s	0.0%
native_flush_tlb_one_user	vmlinux	0.020s	0.0%
[Others]	N/A*	0.385s	0.2%

N/A is applied to non-summable metrics.

Figure 4.4: Top Hotsport analysis for Matrix Multiplication

Intel's VTune profiler also provides us with hotspot analysis. Hotspots are those functions that take a significant amount of CPU time 4.4. This analysis will help us identify and optimize these functions so that we can have a significant improvement in the performance of our application.

There are a few other features of Intel's VTune that are useful. Hardware events, Memory usage, System overview, etc.

4.3 Chapter Summary

In this chapter, we discussed a methodology known as the Top-Down methodology. We saw how the pipeline of modern CPUs is divided into four main categories: Front-End, Back-End, Retiring, and Bad Speculation. We saw how VTune uses this methodology of Top-Down analysis to find the performance bottlenecks.

Chapter 5

Inside the Core: Microarchitecture and Intel Skylake

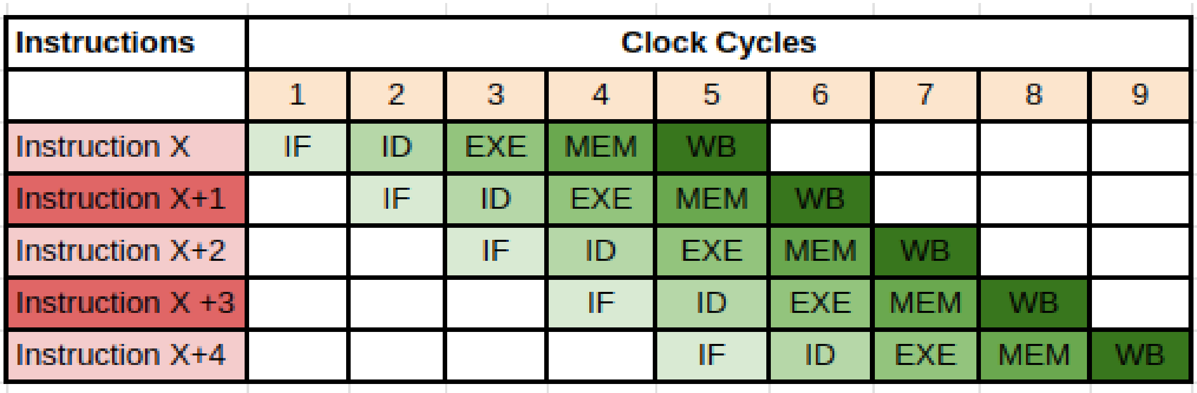
In this chapter, we will discuss the micro-architecture of a processor. Then, we will go into the pipeline of a modern CPU to examine the internal workings of the Intel Skylake microarchitecture. We will also put up the preliminary conditions and setup for our investigations and experiments.

5.1 Microarchitecture of a Processor

Computer architecture is a broad field encompassing both microarchitecture and instruction set architecture[11].

The instruction set architecture (ISA)[11] is essentially the interface between hardware and software. It defines the set of instructions that a processor can execute. Some well-know ISAs include, **X86, ARM, MIPS.** In addition to instructions, the ISA also specifies addressing modes, operand types and sizes, supported operations, and control flow instructions. Different processors ca implement the same ISA but still differ in internal design. For example, AMD opteron and Intel Core i7 both use X86 ISA, but their internal structures like cache size, ports, and pipelines are different.

Talking about the internal structure, we will see what the microarchitecture of a processor is[11]. Microarchitecture refers to how a processor would implement the ISA. It deals with internal organization and behavior of the hardware components in a system like pipelines, caches, and execution units (ports).



Instructions	**Clock Cycles**								
	1	2	3	4	5	6	7	8	9
Instruction X	IF	ID	EXE	MEM	WB				
Instruction X+1		IF	ID	EXE	MEM	WB			
Instruction X+2			IF	ID	EXE	MEM	WB		
Instruction X +3				IF	ID	EXE	MEM	WB	
Instruction X+4					IF	ID	EXE	MEM	WB

Figure 5.1: A Simple 5-stage pipeline.

Modern processors always break the instruction's execution into multiple stages. A simple pipeline 5-stage pipeline is shown in figure 5.1. The common pipeline stages include:

1. Instruction Fetch.

2. Instruction Decode.

3. Execute.

4. Memory Access.

5. Writeback.

Pipelining enables instruction-level parallelism, leading to improved throughput and efficiency. Parallelism is achieved by overlapping multiple instruction execution stages. Some architecture may introduce additional stages for further efficiency. The pipeline design requires us to be careful in balancing the number of stages, delay in each stage, and dependencies between instructions.

As we discussed before the microarchitecture also defines execution units, which is the core processing unit in the processor. Designing them and integrating them accordingly is also important. The execution units such as Arithmetic Logic Units (ALUs), Floating Point Units (FPUs), Load/Store units, Branch Prediction units, and SIMD execution units are responsible for performing the core operations and calculations within a processor. But there is a trade-off that is during the designing phase we should consider the latency, throughput, power usage and chip area to optimize the overall performance.

5.2 Five-Stage Pipeline and Skylake Microarchitecture

5.2.1 Five-Stage Pipeline

There are five basic phases in the traditional five-stage pipeline. As an illustration, we will consider the RISC architectural pipeline that combines integer ALU, branch, and load-store word operations. The RISC subset's instructions can all be completed in a maximum of five clock cycle [12].
The five phases consists of: 5.1

1. **Instruction fetch stage (IF):** The current instruction is fetched from memory using the Program Counter (PC). After fetching, the PC is updated to point to the next instruction.

2. **Instruction Decode (ID):** The instruction is decoded, and the relevant registers are fetched from the register file. Simultaneously, a comparison is performed for equality checks (useful in conditional branches). If needed, the offset field in the instruction is sign-extended, and the potential branch target is calculated by adding this offset to the incremented PC. In aggressive implementations, the branch target address can be stored in the PC at the end of this stage, provided the branch condition is true.

3. **Execution (EX):** Based on the instruction type, the Arithmetic Logic Unit (ALU) performs operations such as memory address calculation (for load/store), register-register arithmetic, or immediate value computations.

4. **Memory Access (MEM):** Load and store instructions access the memory in this stage.

5. **Write-Back (WB):** In this final stage, the result from the ALU or memory is written back to the register file.

5.2.2 Skylake Microarchitecture

The microarchitecture of modern CPU cores is generally divided into two major parts, known as an in-order front-end, which consists of the Instruction fetch and decode of the instructions into ups, and an out-of-order back-end consisting of the other stages like execution, memory,y and write back stages. For example, figure 5.2 shows a block diagram of the microarchitecture of a CPU core in the Intel Skylake Microarchitecture.

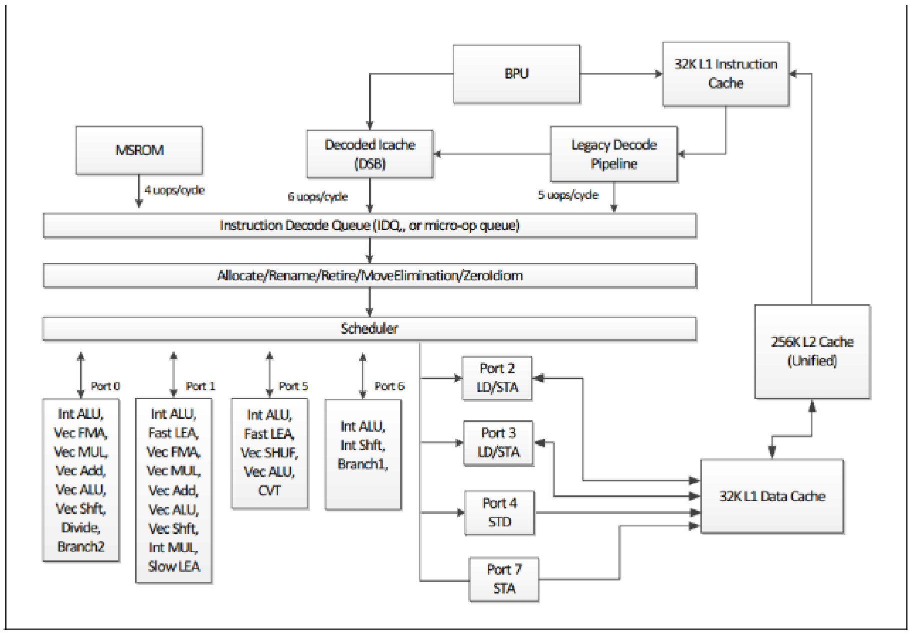

BPU
32K L1 Instruction Cache
MSROM
Decoded Icache (DSB)
Legacy Decode Pipeline
4 uops/cycle
6 uops/cycle
5 uops/cycle
Instruction Decode Queue (IDQ,, or micro-op queue)
Allocate/Rename/Retire/MoveElimination/ZeroIdiom
Scheduler
256K L2 Cache (Unified)
Port 0
Port 1
Port 5
Port 6
Int ALU, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Divide, Branch2
Int ALU, Fast LEA, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Int MUL, Slow LEA
Int ALU, Fast LEA, Vec SHUF, Vec ALU, CVT
Int ALU, Int Shft, Branch1,
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
32K L1 Data Cache

Figure 5.2: CPU Core of Intel Skylake Microarchitecture.

CPU Frontend

The primary objective of the CPU front end is to fetch and decode instructions from memory effectively using various sophisticated data structures. The main purpose of it is to supply the instructions to the CPU back end to do the further processing of the execution of instructions.

The L1 Instruction Cache can deliver 16 bytes of x86 instructions per cycle, which are shared between the two hardware threads. These x86 instructions are complex and variable in length. Therefore, the pre-decode and decode phases of the pipeline translate them into simpler, fixed-length micro-operations (uOps) and place them into the Instruction Decode Queue (IDQ) for the back-end.

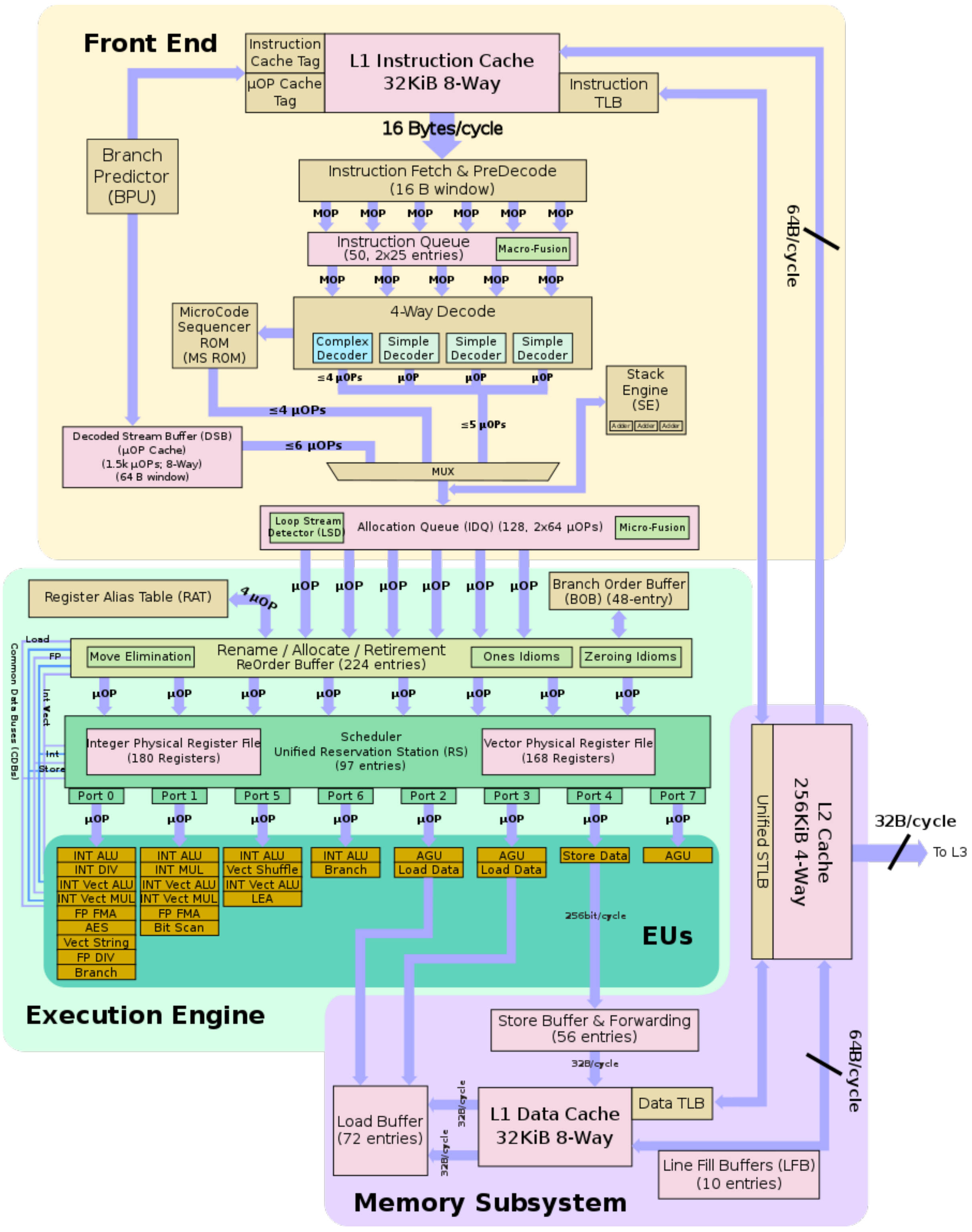
In the pre-decode stage, up to six macro-instructions are fetched and moved into an instruction queue that is split between the two threads. This stage includes support for macro-op fusion, where two x86 instructions can be fused into a single MOP, improving pipeline bandwidth and reducing decode pressure.

Another key structure is the Decoded Stream Buffer (DSB), also known as the UOP Cache. Its function is to cache already-decoded instructions (macro-ops translated to uOps), enabling the front-end to bypass the full decode pipeline when these uOps are re-used. This structure works in parallel with the L1 I-cache and offers up to six uOps per cycle, helping to maintain a balance between the front-end and back-end throughput.

The MicroCode Sequencer ROM (MS ROM) is another structure that helps to handle the instructions that are too complicated to handle. It helps the simple decode unit to handle the complicated instructions.

Additionally, the Branch Prediction Unit (BPU) operates closely with the DSB to ensure that control flow instructions are efficiently handled. The Instruction Decode Queue (IDQ) acts as the interface between the in-order front-end and the out-of-order back-end, ensuring smooth and continuous feeding of instructions into the execution engine.

The Intel Skylake Microarchitecture, including the Front-End, Back-End, and Memory Subsystem, is presented in Figure. 5.3



Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
Branch Predictor (BPU)
16 Bytes/cycle
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
4-Way Decode
Complex Decoder
Simple Decoder
≤4 µOPs
µOP
Stack Engine (SE)
Adder
≤5 µOPs
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
≤6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Move Elimination
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Load
FP
Int Vect
Int
Store
Scheduler Unified Reservation Station (RS) (97 entries)
Integer Physical Register File (180 Registers)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
64B/cycle
Memory Subsystem

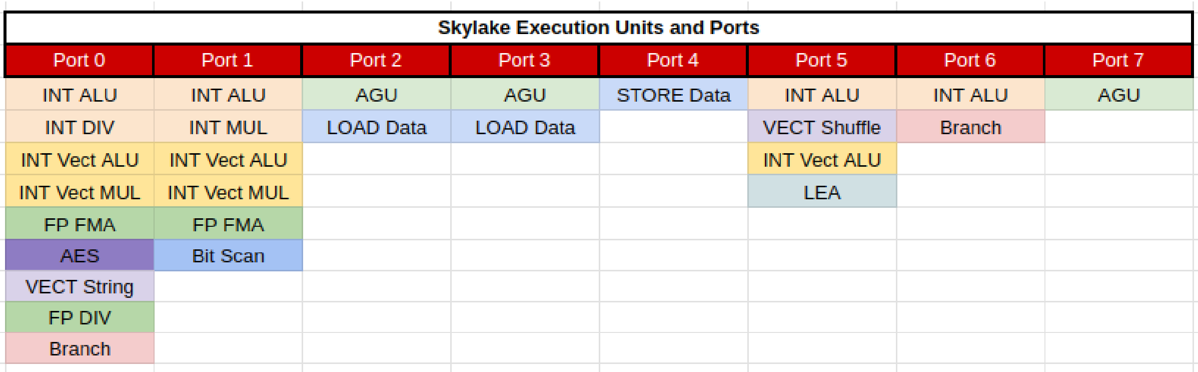
Figure 5.3: Intel Skylake Microarchitecture.

CPU Backend

As we can see above in the figure. 5.3, which illustrates the back end of the Intel Skylake microarchitecture. It is the core part of the pipeline stage when the actual execution of the instructions happens.

Once the front end has allocated the instructions, the back end picks it for Execution. The Back-End employs an Out-of-Order execution to efficiently process instructions and effectively use the pipeline. At the starting lies the Re Order Buffer (ROB), Which in Skylake has 224 entries. The ROB plays an important role in resolving data dependencies, tracking speculative execution, and managing register renaming by linking physical registers used in the Reservation station to architecture-visible registers with the help of a Register alias table (RAT). Even though the execution happens out of order, the retirement of instructions from the ROB always maintains program order.

The Reservation station keeps track of all the resources available and dispatches ready uOps to the corresponding execution ports. The Skylake core is a 4-way super-scalar, which allows the reservation station to issue up to four uOps per cycle.



Skylake Execution Units and Ports

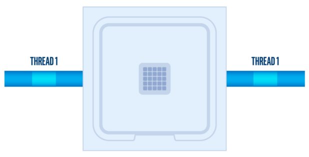
Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
INT ALU	INT ALU	AGU	AGU	STORE Data	INT ALU	INT ALU	AGU
INT DIV	INT MUL	LOAD Data	LOAD Data		VECT Shuffle	Branch	
INT Vect ALU	INT Vect ALU				INT Vect ALU		
INT Vect MUL	INT Vect MUL				LEA		
FP FMA	FP FMA						
AES	Bit Scan						
VECT String							
FP DIV							
Branch							

Figure 5.4: Intel Skylake Microarchitecture Ports.

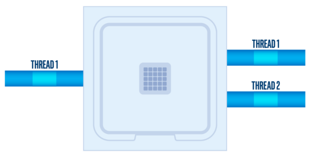
As illustrated in Figure. 5.4 each port is is associated with set of execution ports for different purpose. These ports support a wide range of operations depending on the execution port in that port.

5.3 Intel® Hyper-Threading Technology

Intel's hyper-threading technology is a hardware-level innovation that enables a single physical core to execute multiple threads simultaneously. Each physical core is divided into two logical cores, effectively doubling the number of threads that can manage in parallel.



(a) A physical core without Hyper-Threading (one thread per core)



(b) A physical core with Hyper-Threading enabled (two threads per core)

Figure 5.5: Illustration of Intel® Hyper-Threading Technology

The illustration of Hyper-threading enabled and Hyper-threading disabled is in Figure. 5.5.

When hyper-threading is active, the CPU has two execution instances per physical core. Aa core will act like two different logical cores, where each of them will be capable of handling different independent operations.

We will need to know that all of them will share the same Core execution resources, like Arthematic Logic units (ALUs), Floating Point units (FPUs), and caches.

In this way, sharing of resources can happen, leading to better utilization of them. During the ideal period, if some process is doing memory access or I/O operation, the other process can use the execution units to improve the overall CPU throughput and overall efficiency.

Latency handling is also better in a hyperthreading-enabled CPU than in a Hyperthreading disabled CPU as it does via thread switching.

5.4 Preliminary Experimental Setup

The experimental setup used for the project's experiments is presented in this section. There are numerous machine-learning frameworks available, including TensorFlow, PyTorch, JAX, and others. I have selected the PyTorch framework for my experiments since it is one of the most well-known frameworks in the Deep Learning space and because the experiments are focused on the model's general architecture rather than the frameworks themselves. I used PyTorch version 1.13.1 for this paper. I have utilized the Intel VTuneTM Profiler 2024.0.0 version for these purposes.

With an Intel i5-8350U CPU running at 1.70GHz, the Lenovo Thinkpad T480 serves as the hardware platform. The processor features four physical cores and the Intel Skylake microarchitecture. Because Intel's hyperthreading technology is used, each physical core has two logical cores correspondingly. The CPU is made up of 128KB L1 data and L1 instruction cache. Each core of it has a 1MB L2 cache. It has an 8MB shared L3 cache. Our experimental platform is running Pop! OS 22.04 with a Linux 6.6.6-76060606-generic kernel.

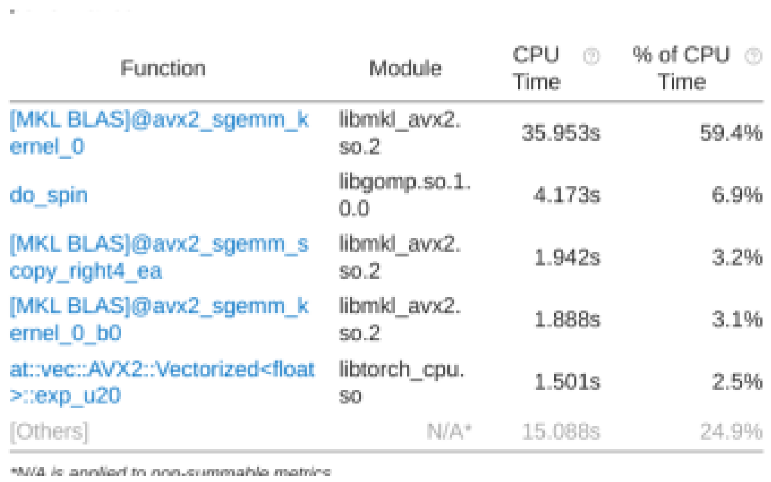
Our experiment deals with Google's BERT Base Uncased model. It is a transformer-based model. BERT Base Uncased was pre-trained on a massive corpus of text data known as BooksCorpus and English Wikipedia. This dataset consists of a diverse collection of text, including books, articles, and other written materials. By training on this extensive dataset, BERT Base Uncased learned to represent language in a way that captures complex contextual relationships between words. 512 Tokens were given as input. The tokenization is done based on WordPiece tokenization. The Pre-trained BERT Base Uncased model outputs the hidden states of each input token and pooled output which is an aggregate representation of the whole sequence.

All the experiment setup and model configuration details are shown in table below:

Parameter	**Value**
Processor	Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz
OS	Pop!_OS 22.04
Profiler	Intel VTune Profiler 2024.0.0
Framework	PyTorch
Model	BERT Base Uncased
Parameters	110M
Optimizer	AdamW
Number of Tokens (Input)	512
Hidden States	768

Table 5.1: System and Model Details

As discussed in the previous section, hyperthreading is a technology that enables a single physical core to function as two logical cores, aiming to improve CPU utilization by allowing multiple threads to share the execution resources of that core. However, when hyperthreading is enabled, logical threads compete for the same physical core resources, leading to contention and delays, particularly during synchronization, where threads wait for each other to reach a certain point. Hence, general matrix multiplication will be bottlenecked.



Function	Module	CPU Time	% of CPU Time
[MKL BLAS]@avx2_sgemm_kernel_0	libmkl_avx2.so.2	35.953s	59.4%
do_spin	libgomp.so.1.0.0	4.173s	6.9%
[MKL BLAS]@avx2_sgemm_scopy_right4_ea	libmkl_avx2.so.2	1.942s	3.2%
[MKL BLAS]@avx2_sgemm_kernel_0_b0	libmkl_avx2.so.2	1.888s	3.1%
at::vec::AVX2::Vectorized<float>::exp_u20	libtorch_cpu.so	1.501s	2.5%
[Others]	N/A*	15.088s	24.9%

*N/A is applied to non-summable metrics

Figure 5.6: VTune Hotspot Analysis with Hyper-Threading Enabled

Hotspot analysis of a deep learning workload with hyperthreading enabled was used even in my initial experiment to confirm the above. It can be observed that the `do_spin` function implements busy-waiting, a technique where a thread repeatedly checks for a condition instead of yielding control, clearly indicating that it is one of those hotspot functions that take a significant amount of CPU time.

Since our goal is to characterize the deep learning workload-related functions rather than solve the synchronization problem, we mitigate these delays by disabling hyperthreading in all of our experiments and only using single threads by setting `torch.set_num_threads = 1`. for all of our experiments.

Hotspot analysis with hyper-threading enabled is illustrated in Figure. 5.6.

5.5 Chapter Summary

In this chapter, we saw what computer micro-architecture is. Then, we discussed the basic five-stage pipeline and how it improves the throughput and efficiency. After that,t we saw these concepts for Intel's Skylake processor. We concluded this section by summarizing the required steps and setup for our experimental analysis.

In the following chapter, we present the relevant experimental procedures undertaken during the project, followed by an examination of the intermediate results.

Chapter 6

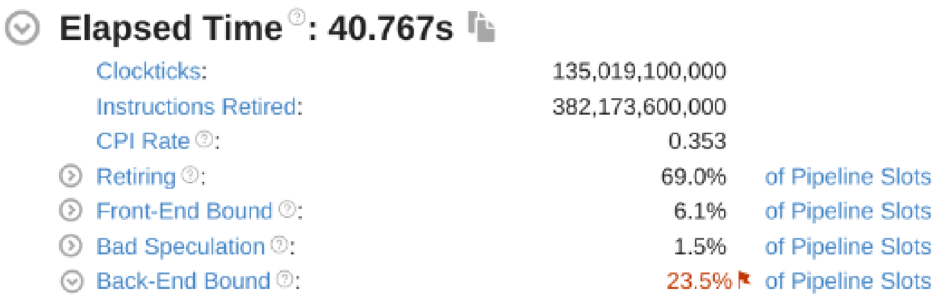
Related Work and Results

In this chapter, we will run our workload, BERT Base Uncased model on VTune. We primarily will focus on the microarchitectural exploration and hotspot analysis. Then, we will try to find if there are any bottlenecks at the hardware or at the software level and check if there are optimization opportunities.

6.1 Top-Down Performance analysis of BERT Base Uncased Model

As we discussed in section 4.1 each pipeline slot available in application runtime is classified into one of the four categories - Front-end, Back-end, Bad-Speculation, Retiring[1]. In the same way, the micro-architecture Exploration of the BERT Base Uncased model's inference is provided below in Figure. 6.1.

As discussed above It is observed that 69.0% of all pipeline slots are retiring, 6.1% are front-end bound, 1.5% are Bad Speculation, and 23.5% of all pipeline slots are back-end bound.

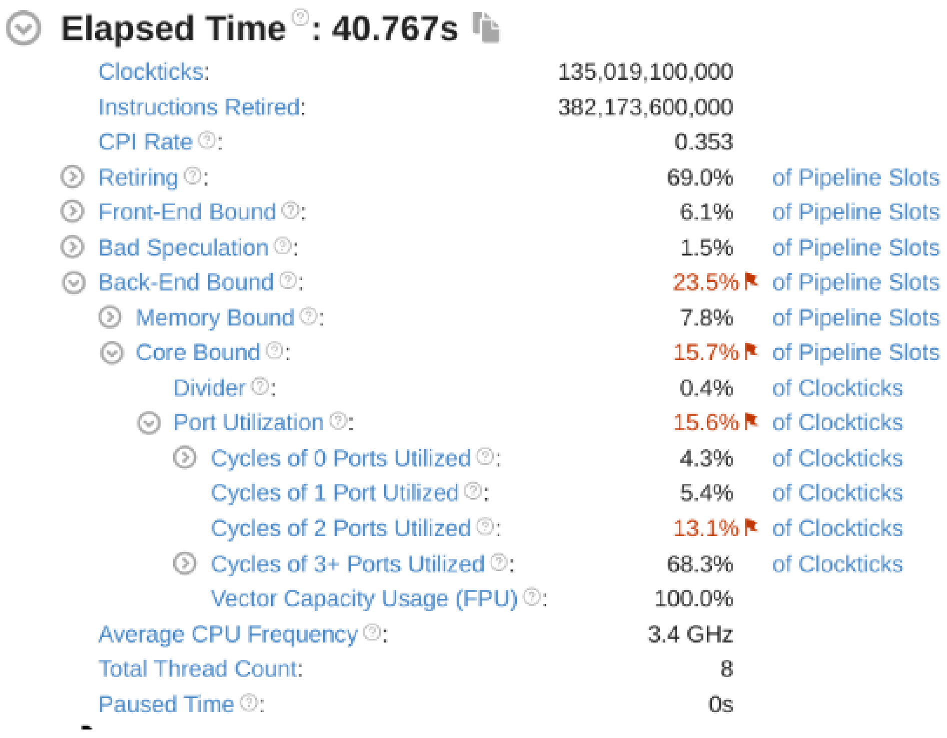


Elapsed Time: 40.767s		
Clockticks:	135,019,100,000	
Instructions Retired:	382,173,600,000	
CPI Rate:	0.353	
Retiring:	69.0%	of Pipeline Slots
Front-End Bound:	6.1%	of Pipeline Slots
Bad Speculation:	1.5%	of Pipeline Slots
Back-End Bound:	23.5%	of Pipeline Slots

Figure 6.1: VTune Microarchitecture analysis Result - 1

So, if we see the result in the above figure. We have the backend-bound category being emphasized. This suggests that we should delve more into the backend-bound category[1]. And can safely ignore the other category.

Further, we drill down into the backend-bound category to find out in the backend-bound category, which categories are highlighted. The two primary subcategories of the backend bound category are Memory bound and Core bound[1]. As we can see in the figure below, core-bound is highlighted between the two sub-categories, suggesting that further investigation into this area is necessary.



Elapsed Time: 40.767s		
Clockticks:	135,019,100,000	
Instructions Retired:	382,173,600,000	
CPI Rate:	0.353	
Retiring:	69.0%	of Pipeline Slots
Front-End Bound:	6.1%	of Pipeline Slots
Bad Speculation:	1.5%	of Pipeline Slots
Back-End Bound:	23.5%	of Pipeline Slots
Memory Bound:	7.8%	of Pipeline Slots
Core Bound:	15.7%	of Pipeline Slots
Divider:	0.4%	of Clockticks
Port Utilization:	15.6%	of Clockticks
Cycles of 0 Ports Utilized:	4.3%	of Clockticks
Cycles of 1 Port Utilized:	5.4%	of Clockticks
Cycles of 2 Ports Utilized:	13.1%	of Clockticks
Cycles of 3+ Ports Utilized:	68.3%	of Clockticks
Vector Capacity Usage (FPU):	100.0%	
Average CPU Frequency:	3.4 GHz	
Total Thread Count:	8	
Paused Time:	0s	

Figure 6.2: VTune Microarchitecture analysis Result - 2

As seen in the figure. 6.2, the port utilization section is flagged among the core-bound categories. The port utilization is a metric that shows the fraction of cycles in which an application stalled due to non-divider-related core difficulties. A subcategory of port utilization, cycles of 2 ports utilized, is the most prominent, accounting for 13.1% of total clock ticks.

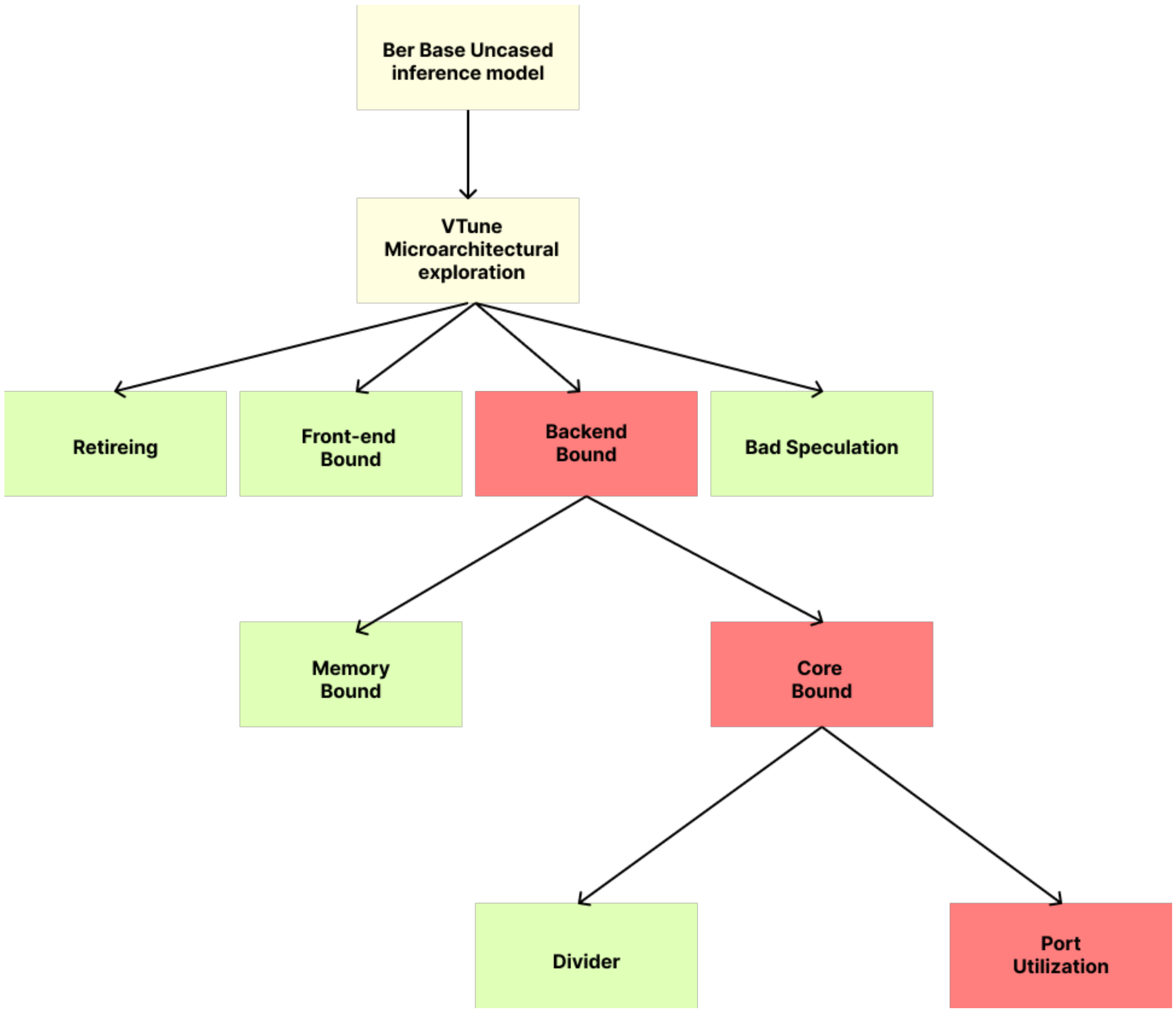


Figure 6.3: Top Down Approach: Summary of VTune Analysis

Port utilization accounts for 15.6% of clock ticks. From the result that we have got there isn't a problem with the divider part, which typically handles the application program's divider and square root portion. Also, we can see that the vector capacity usage(FPU) is 100%. This indicates that the floating point instructions in our program are vectorized at their full capacity. Hence, we can ignore the divider and vector capacity usage sections from our analysis and focus only on the flagged one, which is the cycles of 2 ports utilized.

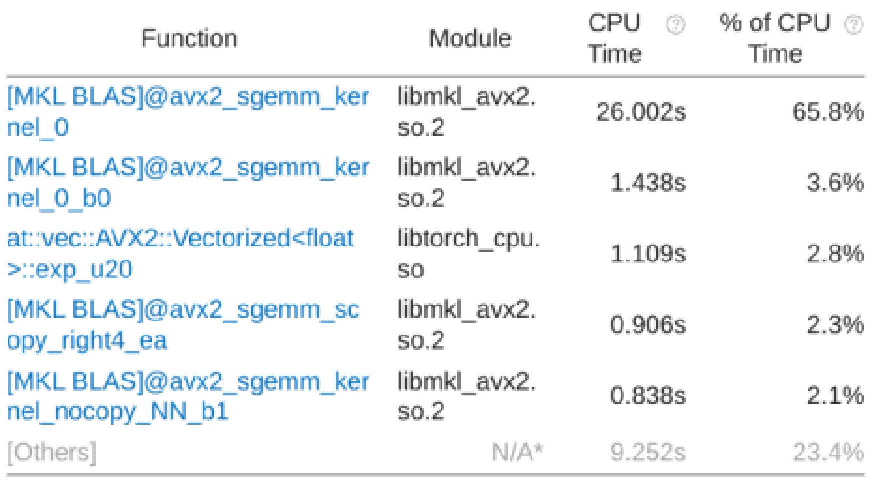
The summary of the microarchitecture exploration for our BERT Base Uncased model is given in Figure. 6.3.

6.2 Hotspot Loop Identification

Optimizing the computational efficiency requires finding the key function and loops that consume the most processing time.

6.2.1 Identifying the Most Active Function

The Figure. 6.4 indicates the most active function in our application.



Function	Module	CPU Time	% of CPU Time
[MKL BLAS]@avx2_sgemm_kernel_0	libmkl_avx2.so.2	26.002s	65.8%
[MKL BLAS]@avx2_sgemm_kernel_0_b0	libmkl_avx2.so.2	1.438s	3.6%
at::vec::AVX2::Vectorized<float>::exp_u20	libtorch_cpu.so	1.109s	2.8%
[MKL BLAS]@avx2_sgemm_scopy_right4_ea	libmkl_avx2.so.2	0.906s	2.3%
[MKL BLAS]@avx2_sgemm_kernel_nocopy_NN_b1	libmkl_avx2.so.2	0.838s	2.1%
[Others]	N/A*	9.252s	23.4%

Figure 6.4: Hotspot analysis: Summary of VTune Analysis

The first function is the one that takes a significant amount of CPU time while running the BERT base uncased Model.

The `MKL BLAS@avx2_sgemm_kernel_0` function belongs to the Intel MKL BLAS library (`libmkl_avx2.so.2`), which leverages the AVX2 (Advanced Vector Extensions 2) instruction set for SIMD (Single Instruction, Multiple Data) operations. The function invoked here is `sgemm_kernel_0`, where SGEMM stands for Single-precision General Matrix Multiply. The `kernel_0` denotes a specific variant of the SGEMM kernel that is highly optimized for the underlying CPU microarchitecture to achieve better performance in matrix multiplication workloads.

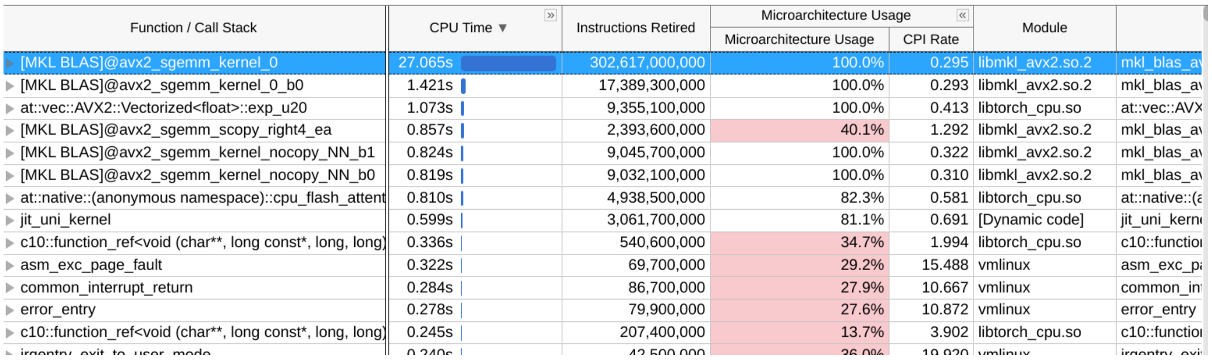
The operation and its details are given below:

$$C := \alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta \cdot C \tag{6.1}$$

where:

- $\mathrm{op}(X)$ is one of $\mathrm{op}(X) = X$, or $\mathrm{op}(X) = X^T$, or $\mathrm{op}(X) = X^H$,
- α and β are scalars,
- A, B, and C are matrices:
  - $\mathrm{op}(A)$ is an $m \times k$ matrix,
  - $\mathrm{op}(B)$ is a $k \times n$ matrix,
  - C is an $m \times n$ matrix.

Now that we have seen how the top hotspot function works, in Figure. 6.5, we see that it is the top part of the VTune Bottom-up tab.

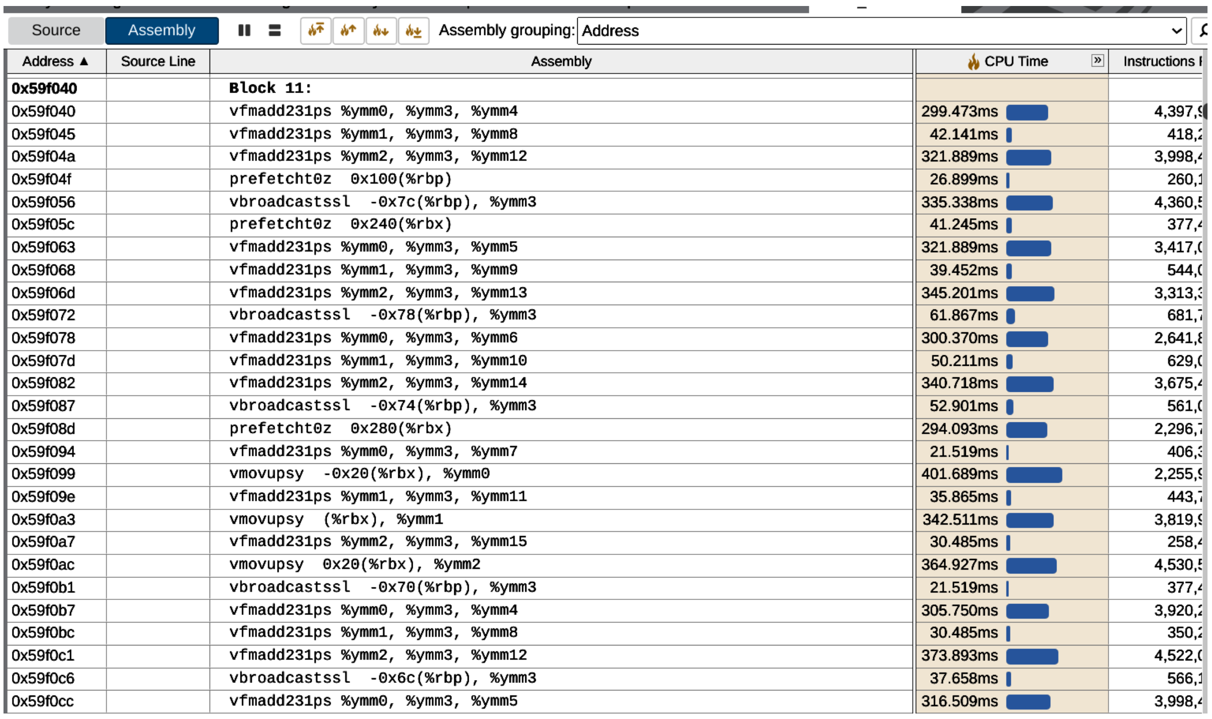


Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage		Module	
			Microarchitecture Usage	CPI Rate		
[MKL BLAS]@avx2_sgemm_kernel_0	27.065s	302,617,000,000	100.0%	0.295	libmkl_avx2.so.2	mkl_blas_av
[MKL BLAS]@avx2_sgemm_kernel_0_b0	1.421s	17,389,300,000	100.0%	0.293	libmkl_avx2.so.2	mkl_blas_av
at::vec::AVX2::Vectorized<float>::exp_u20	1.073s	9,355,100,000	100.0%	0.413	libtorch_cpu.so	at::vec::AVX
[MKL BLAS]@avx2_sgemm_scopy_right4_ea	0.857s	2,393,600,000	40.1%	1.292	libmkl_avx2.so.2	mkl_blas_av
[MKL BLAS]@avx2_sgemm_kernel_nocopy_NN_b1	0.824s	9,045,700,000	100.0%	0.322	libmkl_avx2.so.2	mkl_blas_av
[MKL BLAS]@avx2_sgemm_kernel_nocopy_NN_b0	0.819s	9,032,100,000	100.0%	0.310	libmkl_avx2.so.2	mkl_blas_av
at::native::(anonymous namespace)::cpu_flash_attent	0.810s	4,938,500,000	82.3%	0.581	libtorch_cpu.so	at::native::(a
jit_uni_kernel	0.599s	3,061,700,000	81.1%	0.691	[Dynamic code]	jit_uni_kern
c10::function_ref<void (char**, long const*, long, long)	0.336s	540,600,000	34.7%	1.994	libtorch_cpu.so	c10::functio
asm_exc_page_fault	0.322s	69,700,000	29.2%	15.488	vmlinux	asm_exc_pa
common_interrupt_return	0.284s	86,700,000	27.9%	10.667	vmlinux	common_in
error_entry	0.278s	79,900,000	27.6%	10.872	vmlinux	error_entry
c10::function_ref<void (char**, long const*, long, long)	0.245s	207,400,000	13.7%	3.902	libtorch_cpu.so	c10::functio
irqentry_exit_to_user_mode	0.240s	42,500,000	36.0%	19.920	vmlinux	irqentry_exi

Figure 6.5: Hotspot analysis: Bottom up tab of VTune Analysis

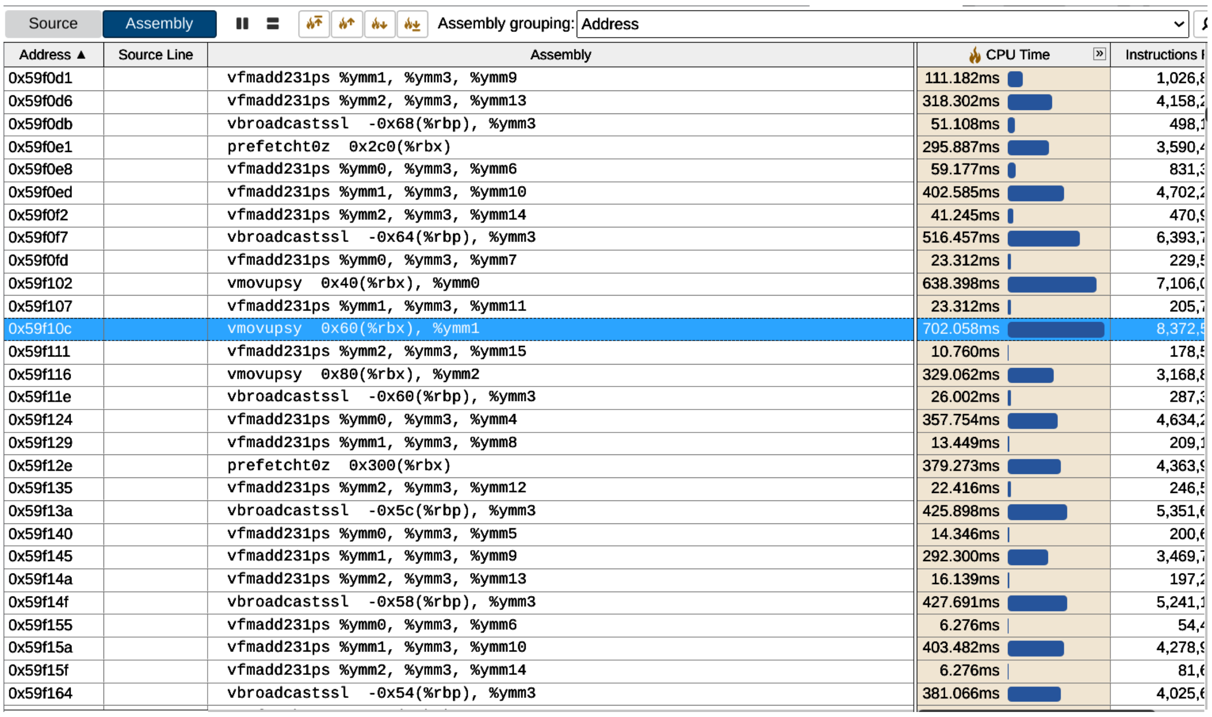
Knowing this, we can delve deeper to understand what this function contains, including any inner functions and loops that contribute to its performance cost. In Figures. 6.6, 6.7, 6.8, 6.9.

The loop below is the one that contributes the most to the hotspot function. That is, it accounts to 68.70% of CPU time of the hotspot function. This gives us further indication to investigate the loop.


Source | Assembly | Assembly grouping: Address

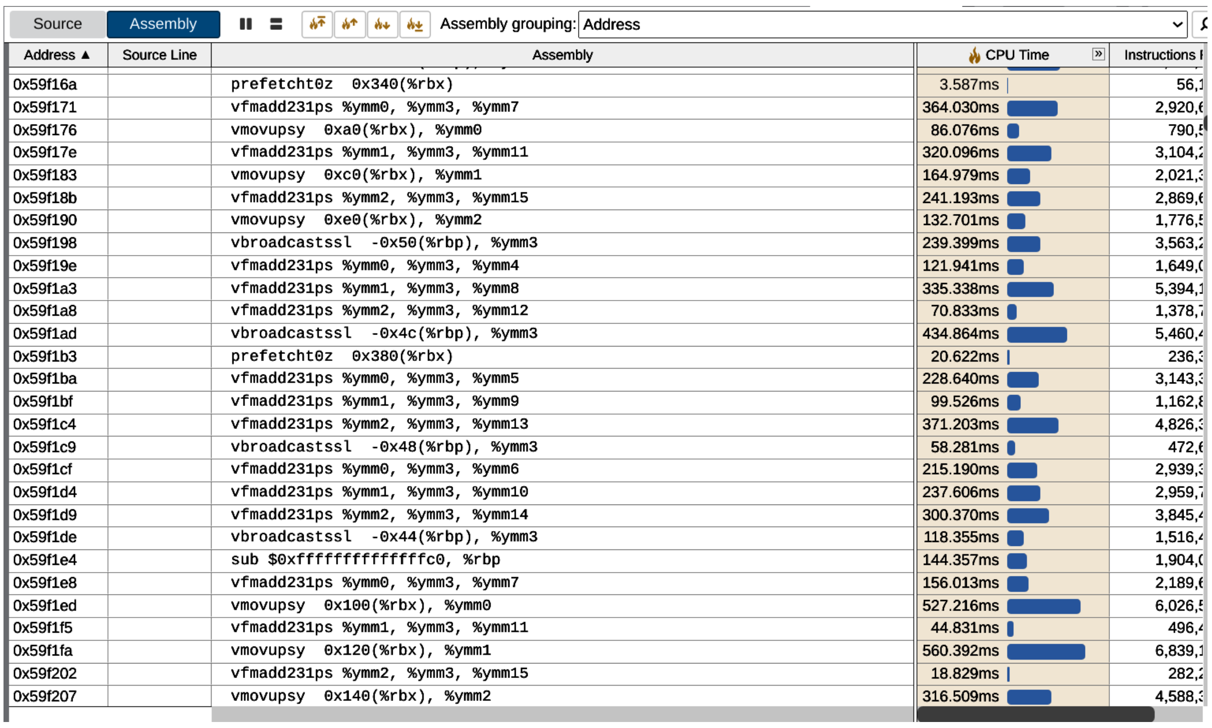
Address ▲	Source Line	Assembly	CPU Time	Instructions
0x59f040		**Block 11:**		
0x59f040		`vfmadd231ps %ymm0, %ymm3, %ymm4`	299.473ms	4,397,9
0x59f045		`vfmadd231ps %ymm1, %ymm3, %ymm8`	42.141ms	418,2
0x59f04a		`vfmadd231ps %ymm2, %ymm3, %ymm12`	321.889ms	3,998,4
0x59f04f		`prefetcht0z 0x100(%rbp)`	26.899ms	260,1
0x59f056		`vbroadcastssl -0x7c(%rbp), %ymm3`	335.338ms	4,360,5
0x59f05c		`prefetcht0z 0x240(%rbx)`	41.245ms	377,4
0x59f063		`vfmadd231ps %ymm0, %ymm3, %ymm5`	321.889ms	3,417,0
0x59f068		`vfmadd231ps %ymm1, %ymm3, %ymm9`	39.452ms	544,0
0x59f06d		`vfmadd231ps %ymm2, %ymm3, %ymm13`	345.201ms	3,313,3
0x59f072		`vbroadcastssl -0x78(%rbp), %ymm3`	61.867ms	681,7
0x59f078		`vfmadd231ps %ymm0, %ymm3, %ymm6`	300.370ms	2,641,8
0x59f07d		`vfmadd231ps %ymm1, %ymm3, %ymm10`	50.211ms	629,0
0x59f082		`vfmadd231ps %ymm2, %ymm3, %ymm14`	340.718ms	3,675,4
0x59f087		`vbroadcastssl -0x74(%rbp), %ymm3`	52.901ms	561,0
0x59f08d		`prefetcht0z 0x280(%rbx)`	294.093ms	2,296,7
0x59f094		`vfmadd231ps %ymm0, %ymm3, %ymm7`	21.519ms	406,3
0x59f099		`vmovupsy -0x20(%rbx), %ymm0`	401.689ms	2,255,9
0x59f09e		`vfmadd231ps %ymm1, %ymm3, %ymm11`	35.865ms	443,7
0x59f0a3		`vmovupsy (%rbx), %ymm1`	342.511ms	3,819,9
0x59f0a7		`vfmadd231ps %ymm2, %ymm3, %ymm15`	30.485ms	258,4
0x59f0ac		`vmovupsy 0x20(%rbx), %ymm2`	364.927ms	4,530,5
0x59f0b1		`vbroadcastssl -0x70(%rbp), %ymm3`	21.519ms	377,4
0x59f0b7		`vfmadd231ps %ymm0, %ymm3, %ymm4`	305.750ms	3,920,2
0x59f0bc		`vfmadd231ps %ymm1, %ymm3, %ymm8`	30.485ms	350,2
0x59f0c1		`vfmadd231ps %ymm2, %ymm3, %ymm12`	373.893ms	4,522,0
0x59f0c6		`vbroadcastssl -0x6c(%rbp), %ymm3`	37.658ms	566,1
0x59f0cc		`vfmadd231ps %ymm0, %ymm3, %ymm5`	316.509ms	3,998,4

Figure 6.6: Hotspot analysis: Hot loop pattern 1


Source | Assembly | Assembly grouping: Address

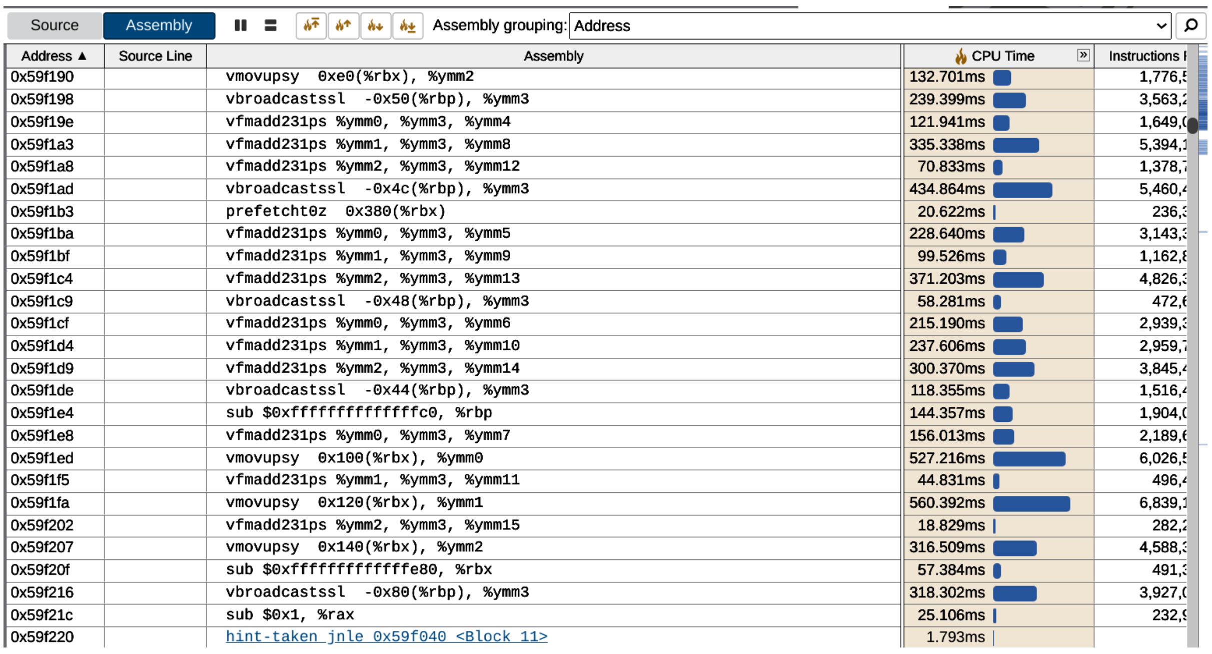
Address ▲	Source Line	Assembly	CPU Time	Instructions
0x59f0d1		`vfmadd231ps %ymm1, %ymm3, %ymm9`	111.182ms	1,026,8
0x59f0d6		`vfmadd231ps %ymm2, %ymm3, %ymm13`	318.302ms	4,158,2
0x59f0db		`vbroadcastssl -0x68(%rbp), %ymm3`	51.108ms	498,1
0x59f0e1		`prefetcht0z 0x2c0(%rbx)`	295.887ms	3,590,4
0x59f0e8		`vfmadd231ps %ymm0, %ymm3, %ymm6`	59.177ms	831,3
0x59f0ed		`vfmadd231ps %ymm1, %ymm3, %ymm10`	402.585ms	4,702,2
0x59f0f2		`vfmadd231ps %ymm2, %ymm3, %ymm14`	41.245ms	470,9
0x59f0f7		`vbroadcastssl -0x64(%rbp), %ymm3`	516.457ms	6,393,7
0x59f0fd		`vfmadd231ps %ymm0, %ymm3, %ymm7`	23.312ms	229,5
0x59f102		`vmovupsy 0x40(%rbx), %ymm0`	638.398ms	7,106,0
0x59f107		`vfmadd231ps %ymm1, %ymm3, %ymm11`	23.312ms	205,7
0x59f10c		`vmovupsy 0x60(%rbx), %ymm1`	702.058ms	8,372,5
0x59f111		`vfmadd231ps %ymm2, %ymm3, %ymm15`	10.760ms	178,5
0x59f116		`vmovupsy 0x80(%rbx), %ymm2`	329.062ms	3,168,8
0x59f11e		`vbroadcastssl -0x60(%rbp), %ymm3`	26.002ms	287,3
0x59f124		`vfmadd231ps %ymm0, %ymm3, %ymm4`	357.754ms	4,634,2
0x59f129		`vfmadd231ps %ymm1, %ymm3, %ymm8`	13.449ms	209,1
0x59f12e		`prefetcht0z 0x300(%rbx)`	379.273ms	4,363,9
0x59f135		`vfmadd231ps %ymm2, %ymm3, %ymm12`	22.416ms	246,5
0x59f13a		`vbroadcastssl -0x5c(%rbp), %ymm3`	425.898ms	5,351,6
0x59f140		`vfmadd231ps %ymm0, %ymm3, %ymm5`	14.346ms	200,6
0x59f145		`vfmadd231ps %ymm1, %ymm3, %ymm9`	292.300ms	3,469,7
0x59f14a		`vfmadd231ps %ymm2, %ymm3, %ymm13`	16.139ms	197,2
0x59f14f		`vbroadcastssl -0x58(%rbp), %ymm3`	427.691ms	5,241,1
0x59f155		`vfmadd231ps %ymm0, %ymm3, %ymm6`	6.276ms	54,4
0x59f15a		`vfmadd231ps %ymm1, %ymm3, %ymm10`	403.482ms	4,278,9
0x59f15f		`vfmadd231ps %ymm2, %ymm3, %ymm14`	6.276ms	81,6
0x59f164		`vbroadcastssl -0x54(%rbp), %ymm3`	381.066ms	4,025,6

Figure 6.7: Hotspot analysis: Hot loop pattern 2



Address ▲	Source Line	Assembly	CPU Time	Instructions I
0x59f16a		`prefetcht0z  0x340(%rbx)`	3.587ms	56,1
0x59f171		`vfmadd231ps %ymm0, %ymm3, %ymm7`	364.030ms	2,920,6
0x59f176		`vmovupsy  0xa0(%rbx), %ymm0`	86.076ms	790,5
0x59f17e		`vfmadd231ps %ymm1, %ymm3, %ymm11`	320.096ms	3,104,2
0x59f183		`vmovupsy  0xc0(%rbx), %ymm1`	164.979ms	2,021,3
0x59f18b		`vfmadd231ps %ymm2, %ymm3, %ymm15`	241.193ms	2,869,6
0x59f190		`vmovupsy  0xe0(%rbx), %ymm2`	132.701ms	1,776,5
0x59f198		`vbroadcastssl  -0x50(%rbp), %ymm3`	239.399ms	3,563,2
0x59f19e		`vfmadd231ps %ymm0, %ymm3, %ymm4`	121.941ms	1,649,0
0x59f1a3		`vfmadd231ps %ymm1, %ymm3, %ymm8`	335.338ms	5,394,1
0x59f1a8		`vfmadd231ps %ymm2, %ymm3, %ymm12`	70.833ms	1,378,7
0x59f1ad		`vbroadcastssl  -0x4c(%rbp), %ymm3`	434.864ms	5,460,4
0x59f1b3		`prefetcht0z  0x380(%rbx)`	20.622ms	236,3
0x59f1ba		`vfmadd231ps %ymm0, %ymm3, %ymm5`	228.640ms	3,143,3
0x59f1bf		`vfmadd231ps %ymm1, %ymm3, %ymm9`	99.526ms	1,162,8
0x59f1c4		`vfmadd231ps %ymm2, %ymm3, %ymm13`	371.203ms	4,826,3
0x59f1c9		`vbroadcastssl  -0x48(%rbp), %ymm3`	58.281ms	472,6
0x59f1cf		`vfmadd231ps %ymm0, %ymm3, %ymm6`	215.190ms	2,939,3
0x59f1d4		`vfmadd231ps %ymm1, %ymm3, %ymm10`	237.606ms	2,959,7
0x59f1d9		`vfmadd231ps %ymm2, %ymm3, %ymm14`	300.370ms	3,845,4
0x59f1de		`vbroadcastssl  -0x44(%rbp), %ymm3`	118.355ms	1,516,4
0x59f1e4		`sub $0xffffffffffffffc0, %rbp`	144.357ms	1,904,0
0x59f1e8		`vfmadd231ps %ymm0, %ymm3, %ymm7`	156.013ms	2,189,6
0x59f1ed		`vmovupsy  0x100(%rbx), %ymm0`	527.216ms	6,026,5
0x59f1f5		`vfmadd231ps %ymm1, %ymm3, %ymm11`	44.831ms	496,4
0x59f1fa		`vmovupsy  0x120(%rbx), %ymm1`	560.392ms	6,839,1
0x59f202		`vfmadd231ps %ymm2, %ymm3, %ymm15`	18.829ms	282,2
0x59f207		`vmovupsy  0x140(%rbx), %ymm2`	316.509ms	4,588,3

Figure 6.8: Hotspot analysis: Hot loop pattern 3



Address ▲	Source Line	Assembly	CPU Time	Instructions I
0x59f190		`vmovupsy  0xe0(%rbx), %ymm2`	132.701ms	1,776,5
0x59f198		`vbroadcastssl  -0x50(%rbp), %ymm3`	239.399ms	3,563,2
0x59f19e		`vfmadd231ps %ymm0, %ymm3, %ymm4`	121.941ms	1,649,0
0x59f1a3		`vfmadd231ps %ymm1, %ymm3, %ymm8`	335.338ms	5,394,1
0x59f1a8		`vfmadd231ps %ymm2, %ymm3, %ymm12`	70.833ms	1,378,7
0x59f1ad		`vbroadcastssl  -0x4c(%rbp), %ymm3`	434.864ms	5,460,4
0x59f1b3		`prefetcht0z  0x380(%rbx)`	20.622ms	236,3
0x59f1ba		`vfmadd231ps %ymm0, %ymm3, %ymm5`	228.640ms	3,143,3
0x59f1bf		`vfmadd231ps %ymm1, %ymm3, %ymm9`	99.526ms	1,162,8
0x59f1c4		`vfmadd231ps %ymm2, %ymm3, %ymm13`	371.203ms	4,826,3
0x59f1c9		`vbroadcastssl  -0x48(%rbp), %ymm3`	58.281ms	472,6
0x59f1cf		`vfmadd231ps %ymm0, %ymm3, %ymm6`	215.190ms	2,939,3
0x59f1d4		`vfmadd231ps %ymm1, %ymm3, %ymm10`	237.606ms	2,959,7
0x59f1d9		`vfmadd231ps %ymm2, %ymm3, %ymm14`	300.370ms	3,845,4
0x59f1de		`vbroadcastssl  -0x44(%rbp), %ymm3`	118.355ms	1,516,4
0x59f1e4		`sub $0xffffffffffffffc0, %rbp`	144.357ms	1,904,0
0x59f1e8		`vfmadd231ps %ymm0, %ymm3, %ymm7`	156.013ms	2,189,6
0x59f1ed		`vmovupsy  0x100(%rbx), %ymm0`	527.216ms	6,026,5
0x59f1f5		`vfmadd231ps %ymm1, %ymm3, %ymm11`	44.831ms	496,4
0x59f1fa		`vmovupsy  0x120(%rbx), %ymm1`	560.392ms	6,839,1
0x59f202		`vfmadd231ps %ymm2, %ymm3, %ymm15`	18.829ms	282,2
0x59f207		`vmovupsy  0x140(%rbx), %ymm2`	316.509ms	4,588,3
0x59f20f		`sub $0xfffffffffffffe80, %rbx`	57.384ms	491,3
0x59f216		`vbroadcastssl  -0x80(%rbp), %ymm3`	318.302ms	3,927,0
0x59f21c		`sub $0x1, %rax`	25.106ms	232,9
0x59f220		`hint-taken jnle 0x59f040 <Block 11>`	1.793ms	

Figure 6.9: Hotspot analysis: Hot loop pattern 4

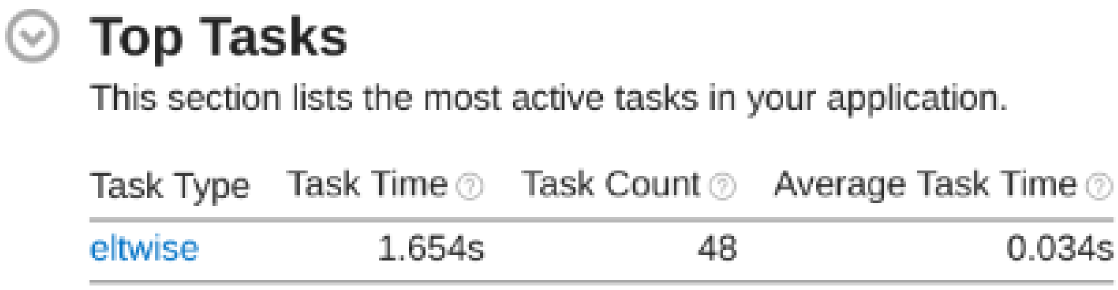
6.2.2 Top Tasks

Eltwise happens to be our top task, where it applies an operation to every element of the tensor[13].

$$\mathrm{dst}_{i_1,\ldots,i_k} = \mathrm{Operation}(\mathrm{src}_{i_1,\ldots,i_k}) \tag{6.2}$$

These are SIMD-friendly operations and are often implemented with AVX instructions in MKL-DNN libraries. Our BERT base uncased model happens to be AVX2 heavy applications.

The element-wise operations are used in the BERT base uncased model in different places:



Top Tasks

This section lists the most active tasks in your application.

Task Type	Task Time	Task Count	Average Task Time
eltwise	1.654s	48	0.034s

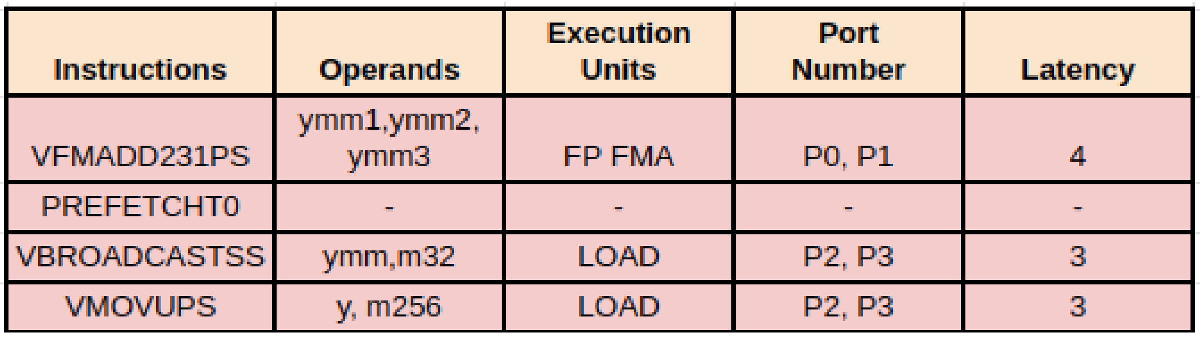
Figure 6.10: Hotspot analysis: Top Tasks

1. **Feed-Forward Layers:** After the attention mechanism is over, the data goes into the feed-forward network, where there is a linear layer, an activation function, and again a linear layer. The activation function used in the BERT base uncased model is the GELU activation function. so it uses `dnnl_eltwise_gelu_tanh`. In the linear layer, we use `dnnl_eltwise_linear` as it is an element-wise operation[13].

2. **Layer Normalization:** After each core sublayer, there is a layer normalization in the encoder blocks of BERT base uncased model, so they use element-wise operations like - `dnnl_eltwise_square`, `dnnl_eltwise_sqrt`, `dnnl_eltwise_linear`[13].

6.3 Identification of Loop Pattern and its relation with Hotspot function

To understand what the loop is consisting, we look at the assembly code of the loop. The assembly code of the loop can also be seen in Figures. 6.6, 6.7, 6.8, 6.9. The assembly code shown in the above figures has few instructions. Now let us understand what these instructions are[14], how they work, and how it is related to the general matrix multiplication, on which is the top hotspot function for our application.



Instructions	Operands	Execution Units	Port Number	Latency
VFMADD231PS	ymm1,ymm2, ymm3	FP FMA	P0, P1	4
PREFETCHT0	-	-	-	-
VBROADCASTSS	ymm,m32	LOAD	P2, P3	3
VMOVUPS	y, m256	LOAD	P2, P3	3

Figure 6.11: Intel Skylake instruction details

The table above illustrates key vector instructions found in performance-critical sections of deep learning workload and in our Hotspot function.

- `vfmadd231ps`: Performs fused multiply-add on packed single-precision floats; utilizes FP FMA units and dispatches on ports `p0/p1`[14], with a latency of 4 cycles this is used in the top hotspot function.
- `prefetcht0`: Hints the CPU to pre-load data into L1 cache. This is not executed like regular instructions, hence, it doesn't consume a port or contribute latency[14].
- `vbroadcastss`: Broadcasts a single scalar value across all elements of a YMM register; utilizes Load units and dispatches through load ports `p2/p3`, with 3-cycle latency[14].
- `vmovups`: Unaligned load of 256-bit YMM register from memory; utilizes Load units and dispatches through `p2/p3`, with 3-cycle latency[14].

Looking at the hotspot function, it is used more in the attention mechanism of our BERT Base uncased model.

6.4 Port Utilization and Its Relation with the Hot Loop

We saw in the section. 6.1 that the cycles of 2 ports are the cause of workload utilization. Now, let us see how this problem relates to the hot loop of our hotspot function. Since there are no Frontend and Backend bound issues, we saw that we could issue up to 4 instructions per cycle from the reservation station to the execution units through ports and retire 4 of them in order. Since the functional units are pipelined, that is we should be able to execute one micro-operation on each stage of a new clock cycle[10].

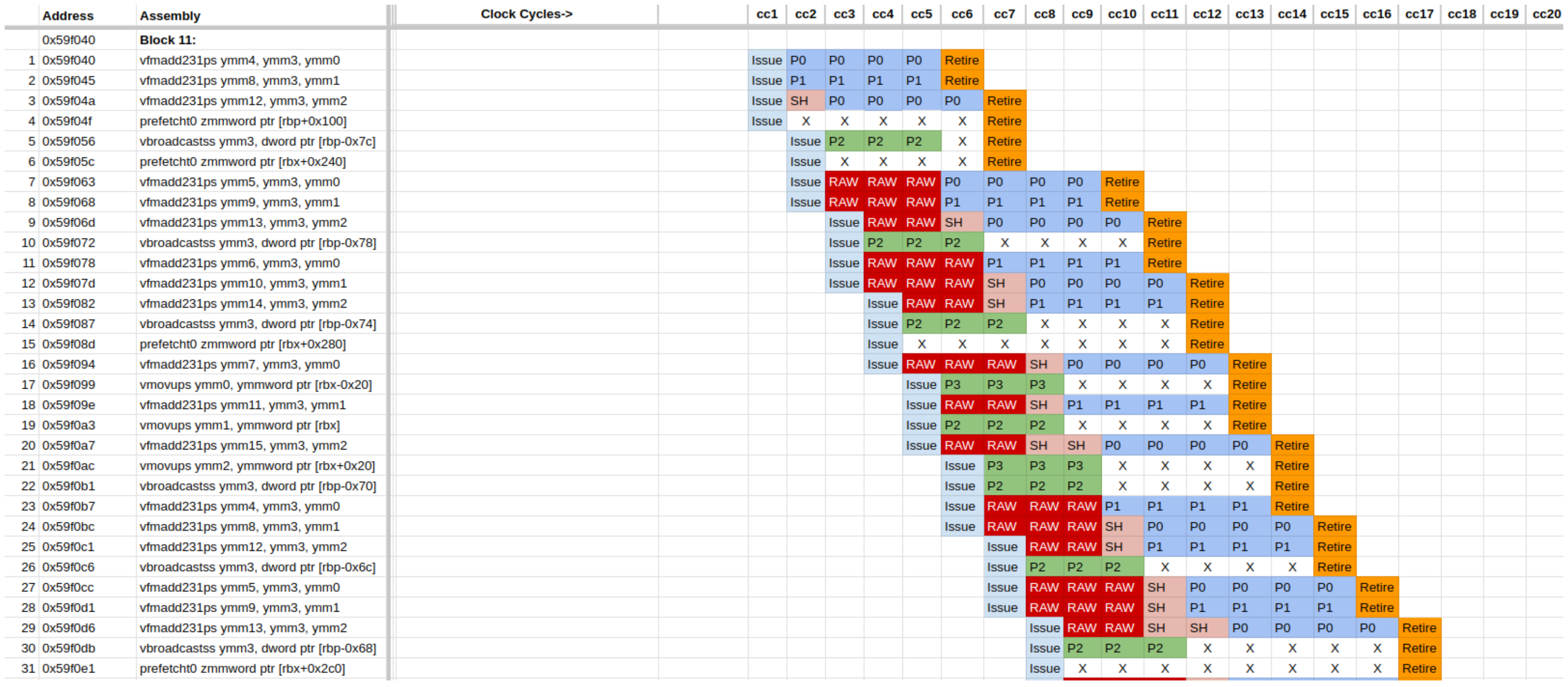
Now, this means that if there should be no problem, we should get cycles of 3+ ports utilized high. Which means that at almost each cycle, we have three or more ports being utilized. Because, as discussed before, we issue 4 instructions, that means that at least 3 ports should be utilized.

But now, in our case, when we run our application that is BERT Base Uncased model, we find that cycles of 2 ports are highlighted[1]. And we have the vector processing unit (VPU) being utilized 100%, which means we should expect cycles of 3+ ports high.

So, there are a few reasons that can lead to this issue:

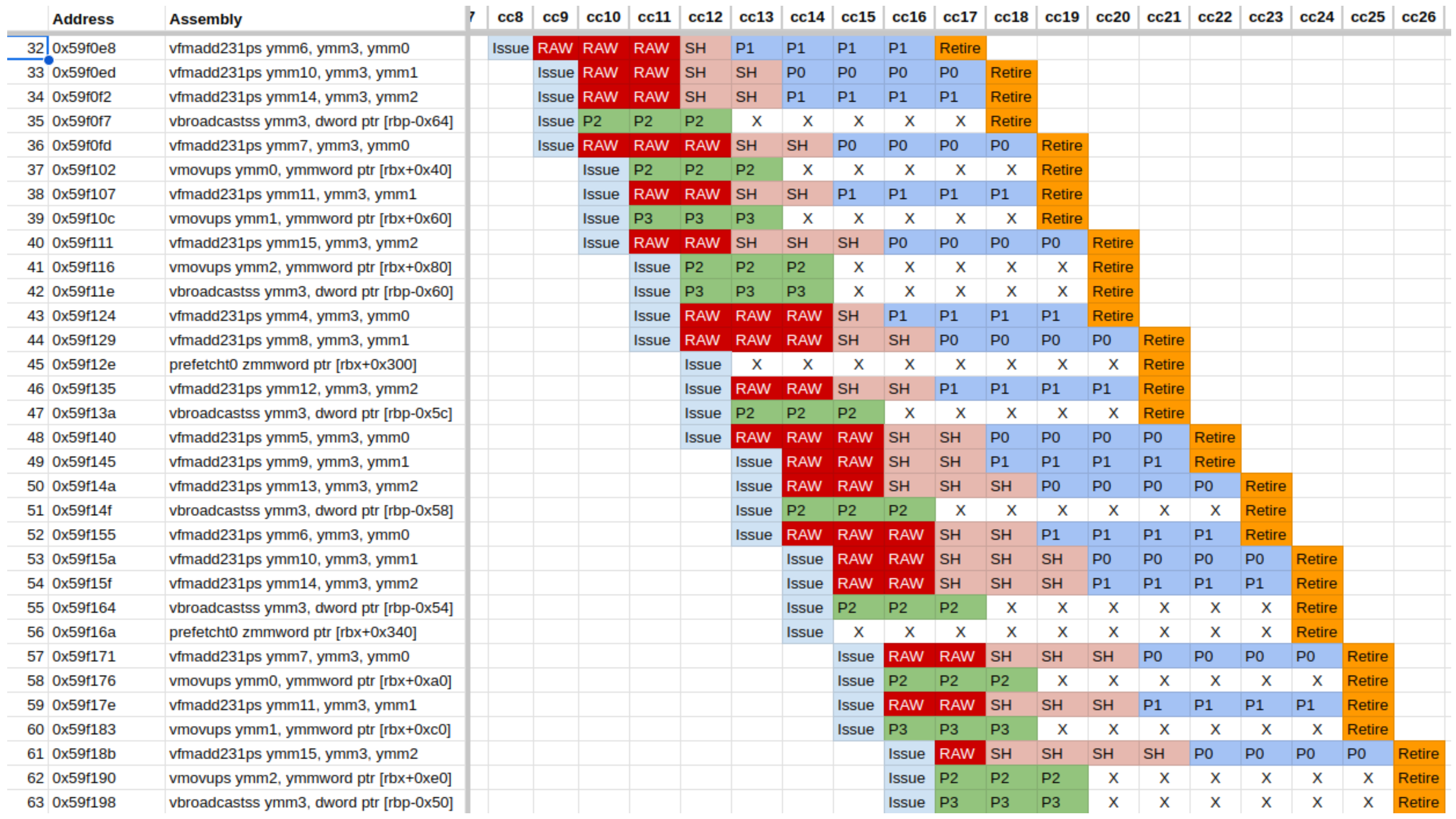
1. **Raw Dependencies**
2. **Structural Hazard:** Insufficient number of execution ports.

So let us see if what we have problem we have discussed above is it causing the cycles of 2 ports getting highlighted. We seen in the 5.4 which execution port is mapped to which execution unit and we have see how instructions work in section. 6.3.



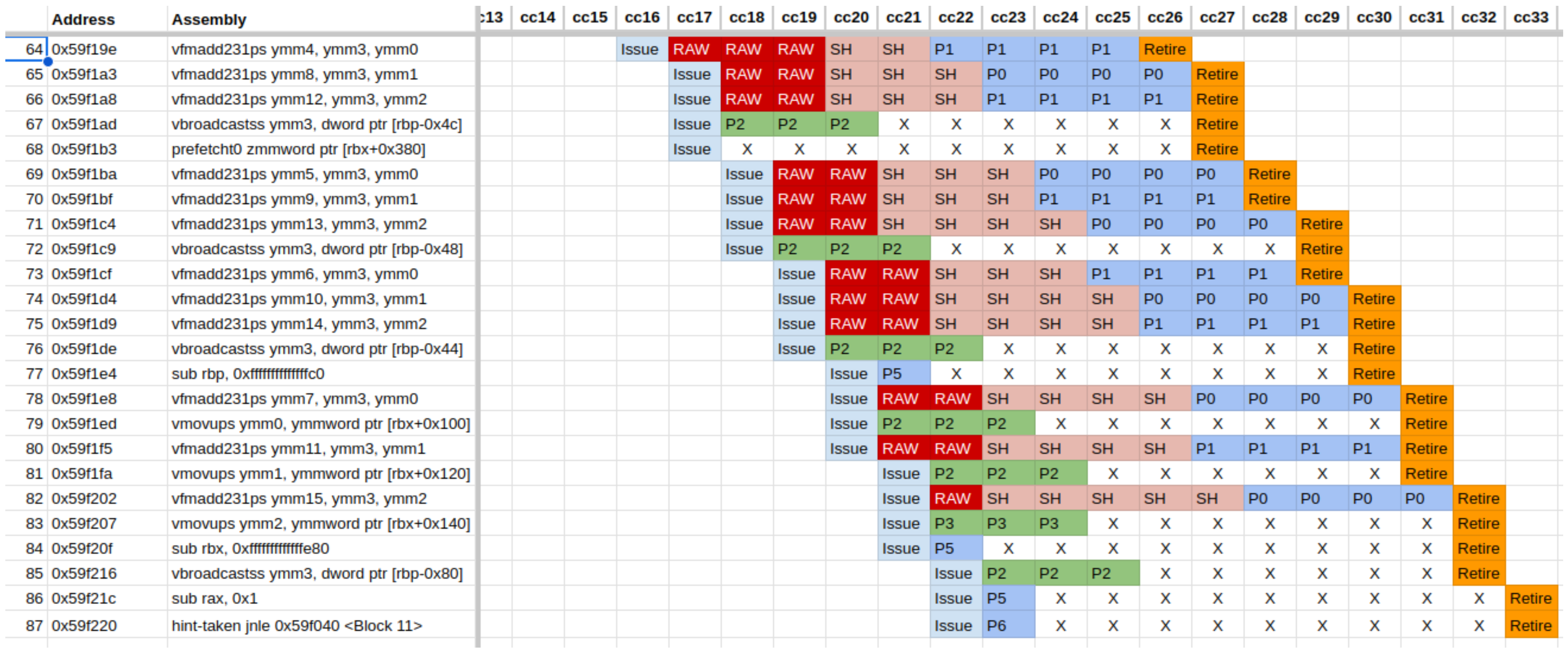
	Address	Assembly	Clock Cycles->	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9	cc10	cc11	cc12	cc13	cc14	cc15	cc16	cc17	cc18	cc19	cc20
	0x59f040	**Block 11:**																					
1	0x59f040	vfmadd231ps ymm4, ymm3, ymm0		Issue	P0	P0	P0	P0	Retire														
2	0x59f045	vfmadd231ps ymm8, ymm3, ymm1		Issue	P1	P1	P1	P1	Retire														
3	0x59f04a	vfmadd231ps ymm12, ymm3, ymm2		Issue	SH	P0	P0	P0	P0	Retire													
4	0x59f04f	prefetcht0 zmmword ptr [rbp+0x100]		Issue	X	X	X	X	X	Retire													
5	0x59f056	vbroadcastss ymm3, dword ptr [rbp-0x7c]			Issue	P2	P2	P2	X	Retire													
6	0x59f05c	prefetcht0 zmmword ptr [rbx+0x240]			Issue	X	X	X	X	Retire													
7	0x59f063	vfmadd231ps ymm5, ymm3, ymm0			Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire										
8	0x59f068	vfmadd231ps ymm9, ymm3, ymm1			Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire										
9	0x59f06d	vfmadd231ps ymm13, ymm3, ymm2				Issue	RAW	RAW	SH	P0	P0	P0	P0	Retire									
10	0x59f072	vbroadcastss ymm3, dword ptr [rbp-0x78]				Issue	P2	P2	P2	X	X	X	X	Retire									
11	0x59f078	vfmadd231ps ymm6, ymm3, ymm0				Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire									
12	0x59f07d	vfmadd231ps ymm10, ymm3, ymm1				Issue	RAW	RAW	RAW	SH	P0	P0	P0	P0	Retire								
13	0x59f082	vfmadd231ps ymm14, ymm3, ymm2					Issue	RAW	RAW	SH	P1	P1	P1	P1	Retire								
14	0x59f087	vbroadcastss ymm3, dword ptr [rbp-0x74]					Issue	P2	P2	P2	X	X	X	X	Retire								
15	0x59f08d	prefetcht0 zmmword ptr [rbx+0x280]					Issue	X	X	X	X	X	X	X	Retire								
16	0x59f094	vfmadd231ps ymm7, ymm3, ymm0					Issue	RAW	RAW	RAW	SH	P0	P0	P0	P0	Retire							
17	0x59f099	vmovups ymm0, ymmword ptr [rbx-0x20]						Issue	P3	P3	P3	X	X	X	X	Retire							
18	0x59f09e	vfmadd231ps ymm11, ymm3, ymm1						Issue	RAW	RAW	SH	P1	P1	P1	P1	Retire							
19	0x59f0a3	vmovups ymm1, ymmword ptr [rbx]						Issue	P2	P2	P2	X	X	X	X	Retire							
20	0x59f0a7	vfmadd231ps ymm15, ymm3, ymm2						Issue	RAW	RAW	SH	SH	P0	P0	P0	P0	Retire						
21	0x59f0ac	vmovups ymm2, ymmword ptr [rbx+0x20]							Issue	P3	P3	P3	X	X	X	X	Retire						
22	0x59f0b1	vbroadcastss ymm3, dword ptr [rbp-0x70]							Issue	P2	P2	P2	X	X	X	X	Retire						
23	0x59f0b7	vfmadd231ps ymm4, ymm3, ymm0							Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire						
24	0x59f0bc	vfmadd231ps ymm8, ymm3, ymm1							Issue	RAW	RAW	RAW	SH	P0	P0	P0	P0	Retire					
25	0x59f0c1	vfmadd231ps ymm12, ymm3, ymm2								Issue	RAW	RAW	SH	P1	P1	P1	P1	Retire					
26	0x59f0c6	vbroadcastss ymm3, dword ptr [rbp-0x6c]								Issue	P2	P2	P2	X	X	X	X	Retire					
27	0x59f0cc	vfmadd231ps ymm5, ymm3, ymm0								Issue	RAW	RAW	RAW	SH	P0	P0	P0	P0	Retire				
28	0x59f0d1	vfmadd231ps ymm9, ymm3, ymm1								Issue	RAW	RAW	RAW	SH	P1	P1	P1	P1	Retire				
29	0x59f0d6	vfmadd231ps ymm13, ymm3, ymm2									Issue	RAW	RAW	SH	SH	P0	P0	P0	P0	Retire			
30	0x59f0db	vbroadcastss ymm3, dword ptr [rbp-0x68]									Issue	P2	P2	P2	X	X	X	X	X	Retire			
31	0x59f0e1	prefetcht0 zmmword ptr [rbx+0x2c0]									Issue	X	X	X	X	X	X	X	X	Retire			

Figure 6.12: Gantt chart of the hot loop instructions 1



	Address	Assembly	cc8	cc9	cc10	cc11	cc12	cc13	cc14	cc15	cc16	cc17	cc18	cc19	cc20	cc21	cc22	cc23	cc24	cc25	cc26
32	0x59f0e8	vfmadd231ps ymm6, ymm3, ymm0	Issue	RAW	RAW	RAW	SH	P1	P1	P1	P1	Retire									
33	0x59f0ed	vfmadd231ps ymm10, ymm3, ymm1		Issue	RAW	RAW	SH	SH	P0	P0	P0	P0	Retire								
34	0x59f0f2	vfmadd231ps ymm14, ymm3, ymm2		Issue	RAW	RAW	SH	SH	P1	P1	P1	P1	Retire								
35	0x59f0f7	vbroadcastss ymm3, dword ptr [rbp-0x64]		Issue	P2	P2	P2	X	X	X	X	X	Retire								
36	0x59f0fd	vfmadd231ps ymm7, ymm3, ymm0		Issue	RAW	RAW	RAW	SH	SH	P0	P0	P0	P0	Retire							
37	0x59f102	vmovups ymm0, ymmword ptr [rbx+0x40]			Issue	P2	P2	P2	X	X	X	X	X	Retire							
38	0x59f107	vfmadd231ps ymm11, ymm3, ymm1			Issue	RAW	RAW	SH	SH	P1	P1	P1	P1	Retire							
39	0x59f10c	vmovups ymm1, ymmword ptr [rbx+0x60]			Issue	P3	P3	P3	X	X	X	X	X	Retire							
40	0x59f111	vfmadd231ps ymm15, ymm3, ymm2			Issue	RAW	RAW	SH	SH	SH	P0	P0	P0	P0	Retire						
41	0x59f116	vmovups ymm2, ymmword ptr [rbx+0x80]				Issue	P2	P2	P2	X	X	X	X	X	Retire						
42	0x59f11e	vbroadcastss ymm3, dword ptr [rbp-0x60]				Issue	P3	P3	P3	X	X	X	X	X	Retire						
43	0x59f124	vfmadd231ps ymm4, ymm3, ymm0				Issue	RAW	RAW	RAW	SH	P1	P1	P1	P1	Retire						
44	0x59f129	vfmadd231ps ymm8, ymm3, ymm1				Issue	RAW	RAW	RAW	SH	SH	P0	P0	P0	P0	Retire					
45	0x59f12e	prefetcht0 zmmword ptr [rbx+0x300]					Issue	X	X	X	X	X	X	X	X	Retire					
46	0x59f135	vfmadd231ps ymm12, ymm3, ymm2					Issue	RAW	RAW	SH	SH	P1	P1	P1	P1	Retire					
47	0x59f13a	vbroadcastss ymm3, dword ptr [rbp-0x5c]					Issue	P2	P2	P2	X	X	X	X	X	Retire					
48	0x59f140	vfmadd231ps ymm5, ymm3, ymm0					Issue	RAW	RAW	RAW	SH	SH	P0	P0	P0	P0	Retire				
49	0x59f145	vfmadd231ps ymm9, ymm3, ymm1						Issue	RAW	RAW	SH	SH	P1	P1	P1	P1	Retire				
50	0x59f14a	vfmadd231ps ymm13, ymm3, ymm2						Issue	RAW	RAW	SH	SH	SH	P0	P0	P0	P0	Retire			
51	0x59f14f	vbroadcastss ymm3, dword ptr [rbp-0x58]						Issue	P2	P2	P2	X	X	X	X	X	X	Retire			
52	0x59f155	vfmadd231ps ymm6, ymm3, ymm0						Issue	RAW	RAW	RAW	SH	SH	P1	P1	P1	P1	Retire			
53	0x59f15a	vfmadd231ps ymm10, ymm3, ymm1							Issue	RAW	RAW	SH	SH	SH	P0	P0	P0	P0	Retire		
54	0x59f15f	vfmadd231ps ymm14, ymm3, ymm2							Issue	RAW	RAW	SH	SH	SH	P1	P1	P1	P1	Retire		
55	0x59f164	vbroadcastss ymm3, dword ptr [rbp-0x54]							Issue	P2	P2	P2	X	X	X	X	X	X	Retire		
56	0x59f16a	prefetcht0 zmmword ptr [rbx+0x340]							Issue	X	X	X	X	X	X	X	X	X	Retire		
57	0x59f171	vfmadd231ps ymm7, ymm3, ymm0								Issue	RAW	RAW	SH	SH	SH	P0	P0	P0	P0	Retire	
58	0x59f176	vmovups ymm0, ymmword ptr [rbx+0xa0]								Issue	P2	P2	P2	X	X	X	X	X	X	Retire	
59	0x59f17e	vfmadd231ps ymm11, ymm3, ymm1								Issue	RAW	RAW	SH	SH	SH	P1	P1	P1	P1	Retire	
60	0x59f183	vmovups ymm1, ymmword ptr [rbx+0xc0]								Issue	P3	P3	P3	X	X	X	X	X	X	Retire	
61	0x59f18b	vfmadd231ps ymm15, ymm3, ymm2									Issue	RAW	SH	SH	SH	SH	P0	P0	P0	P0	Retire
62	0x59f190	vmovups ymm2, ymmword ptr [rbx+0xe0]									Issue	P2	P2	P2	X	X	X	X	X	X	Retire
63	0x59f198	vbroadcastss ymm3, dword ptr [rbp-0x50]									Issue	P3	P3	P3	X	X	X	X	X	X	Retire

Figure 6.13: Gantt chart of the hot loop instructions 2



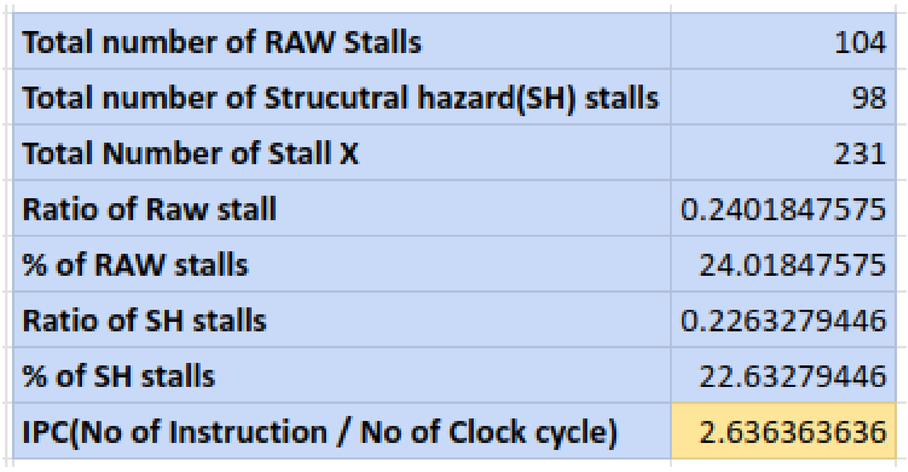
	Address	Assembly	c13	cc14	cc15	cc16	cc17	cc18	cc19	cc20	cc21	cc22	cc23	cc24	cc25	cc26	cc27	cc28	cc29	cc30	cc31	cc32	cc33
64	0x59f19e	vfmadd231ps ymm4, ymm3, ymm0				Issue	RAW	RAW	RAW	SH	SH	P1	P1	P1	P1	Retire							
65	0x59f1a3	vfmadd231ps ymm8, ymm3, ymm1					Issue	RAW	RAW	SH	SH	SH	P0	P0	P0	P0	Retire						
66	0x59f1a8	vfmadd231ps ymm12, ymm3, ymm2					Issue	RAW	RAW	SH	SH	SH	P1	P1	P1	P1	Retire						
67	0x59f1ad	vbroadcastss ymm3, dword ptr [rbp-0x4c]					Issue	P2	P2	P2	X	X	X	X	X	X	Retire						
68	0x59f1b3	prefetcht0 zmmword ptr [rbx+0x380]					Issue	X	X	X	X	X	X	X	X	X	Retire						
69	0x59f1ba	vfmadd231ps ymm5, ymm3, ymm0						Issue	RAW	RAW	SH	SH	SH	P0	P0	P0	P0	Retire					
70	0x59f1bf	vfmadd231ps ymm9, ymm3, ymm1						Issue	RAW	RAW	SH	SH	SH	P1	P1	P1	P1	Retire					
71	0x59f1c4	vfmadd231ps ymm13, ymm3, ymm2						Issue	RAW	RAW	SH	SH	SH	SH	P0	P0	P0	P0	Retire				
72	0x59f1c9	vbroadcastss ymm3, dword ptr [rbp-0x48]						Issue	P2	P2	P2	X	X	X	X	X	X	X	Retire				
73	0x59f1cf	vfmadd231ps ymm6, ymm3, ymm0							Issue	RAW	RAW	SH	SH	SH	P1	P1	P1	P1	Retire				
74	0x59f1d4	vfmadd231ps ymm10, ymm3, ymm1							Issue	RAW	RAW	SH	SH	SH	SH	P0	P0	P0	P0	Retire			
75	0x59f1d9	vfmadd231ps ymm14, ymm3, ymm2							Issue	RAW	RAW	SH	SH	SH	SH	P1	P1	P1	P1	Retire			
76	0x59f1de	vbroadcastss ymm3, dword ptr [rbp-0x44]							Issue	P2	P2	P2	X	X	X	X	X	X	X	Retire			
77	0x59f1e4	sub rbp, 0xffffffffffffffc0								Issue	P5	X	X	X	X	X	X	X	X	Retire			
78	0x59f1e8	vfmadd231ps ymm7, ymm3, ymm0								Issue	RAW	RAW	SH	SH	SH	SH	P0	P0	P0	P0	Retire		
79	0x59f1ed	vmovups ymm0, ymmword ptr [rbx+0x100]								Issue	P2	P2	P2	X	X	X	X	X	X	X	Retire		
80	0x59f1f5	vfmadd231ps ymm11, ymm3, ymm1								Issue	RAW	RAW	SH	SH	SH	SH	P1	P1	P1	P1	Retire		
81	0x59f1fa	vmovups ymm1, ymmword ptr [rbx+0x120]									Issue	P2	P2	P2	X	X	X	X	X	X	Retire		
82	0x59f202	vfmadd231ps ymm15, ymm3, ymm2									Issue	RAW	SH	SH	SH	SH	SH	P0	P0	P0	P0	Retire	
83	0x59f207	vmovups ymm2, ymmword ptr [rbx+0x140]									Issue	P3	P3	P3	X	X	X	X	X	X	X	Retire	
84	0x59f20f	sub rbx, 0xffffffffffffe80									Issue	P5	X	X	X	X	X	X	X	X	X	Retire	
85	0x59f216	vbroadcastss ymm3, dword ptr [rbp-0x80]										Issue	P2	P2	P2	X	X	X	X	X	X	Retire	
86	0x59f21c	sub rax, 0x1										Issue	P5	X	X	X	X	X	X	X	X	X	Retire
87	0x59f220	hint-taken jnle 0x59f040 <Block 11>										Issue	P6	X	X	X	X	X	X	X	X	X	Retire

Figure 6.14: Gantt chart of the hot loop instructions 3

So, using this knowledge, we made the grant chart for the hot loop of our hotspot function, which is illustrated in Figure. 6.12, 6.13, 6.14, the theoretical pipeline view.

The empty slots, which are simple pipeline stalls, are presented by X. The functional unit LOAD is associated with "vmovups" and "vbroadcastss" instructions that use ports p2 and p3. The functional unit FP FMA is associated with the "vfmadd231ps" instruction that uses ports p0 and p1 [14]. The structural hazard caused is represented with "SH" and the RAW dependencies are represented with "RAW".

As discussed earlier, we issued four micro-operations per cycles but only two of them were getting executed in one clock cycle. This happened because of a lack of execution units like FP FMA.



Total number of RAW Stalls	104
Total number of Strucutral hazard(SH) stalls	98
Total Number of Stall X	231
Ratio of Raw stall	0.2401847575
% of RAW stalls	24.01847575
Ratio of SH stalls	0.2263279446
% of SH stalls	22.63279446
IPC(No of Instruction / No of Clock cycle)	2.636363636

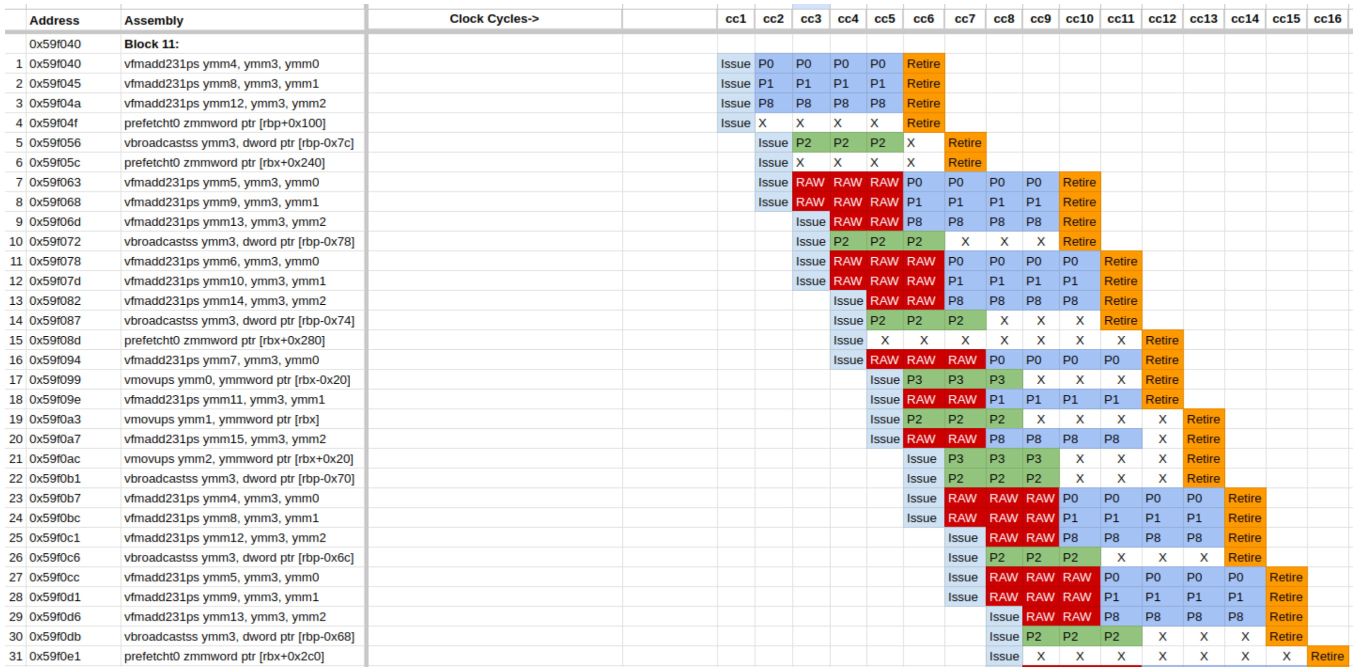
Figure 6.15: Summary of the above Gantt Chart

We also have RAW dependencies, but they cannot be eliminated as they are inherent to the program [12].The figure. 6.15 illustrates the summary of the above Gantt charts.

In the next section, we will try to give a theoretical solution for the problem of an inadequate number of execution units.

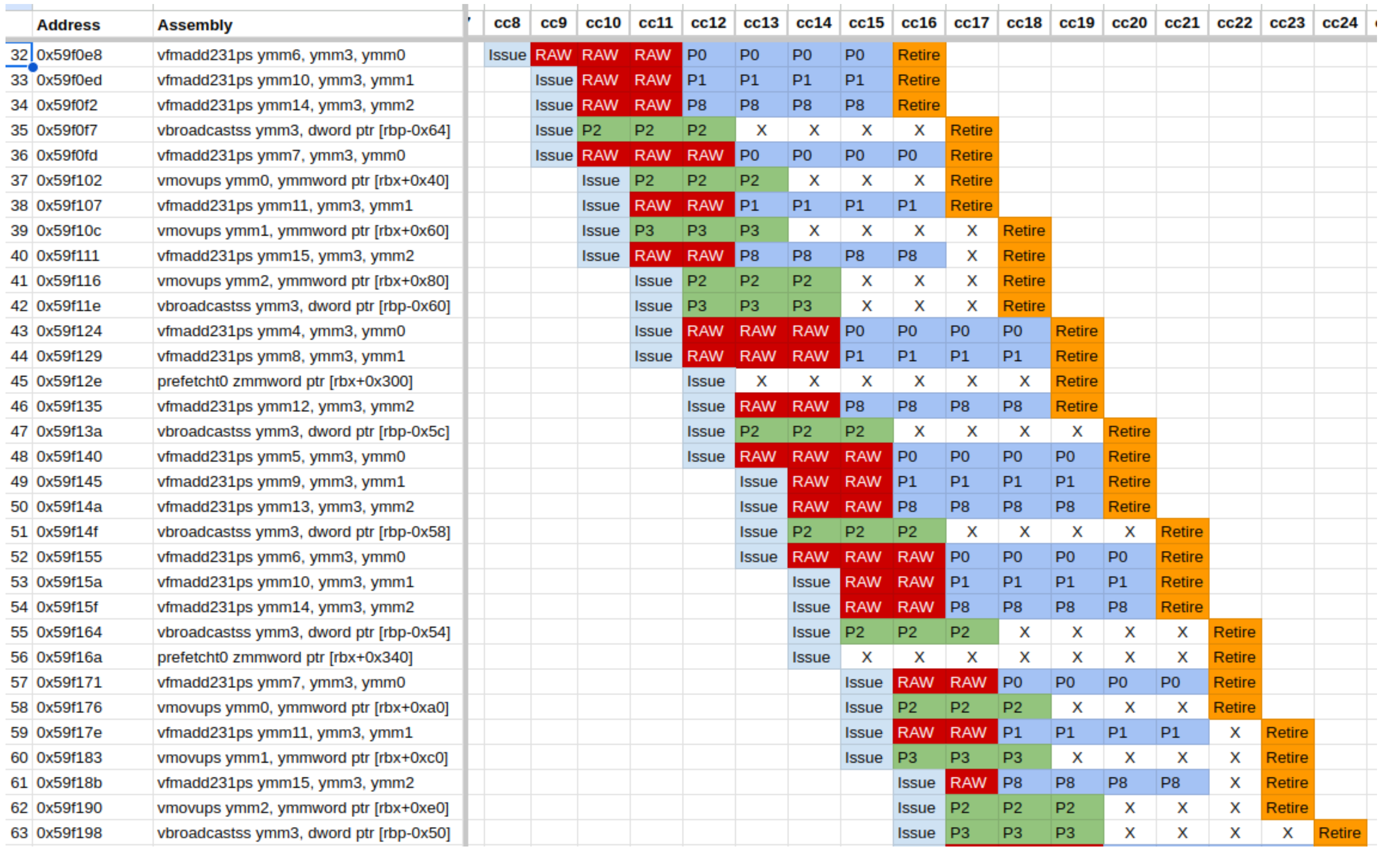
6.5 Microarchitectural improvement suggestion

As we saw the hotloop and made the theoretical pipeline view using Gantt chart. The vfmadd231ps has pipeline slot empty We can see that the structural hazard that is being caused is only because of one execution unit not being there, which is FP FMA, used for floating point fused multiplication and addition.



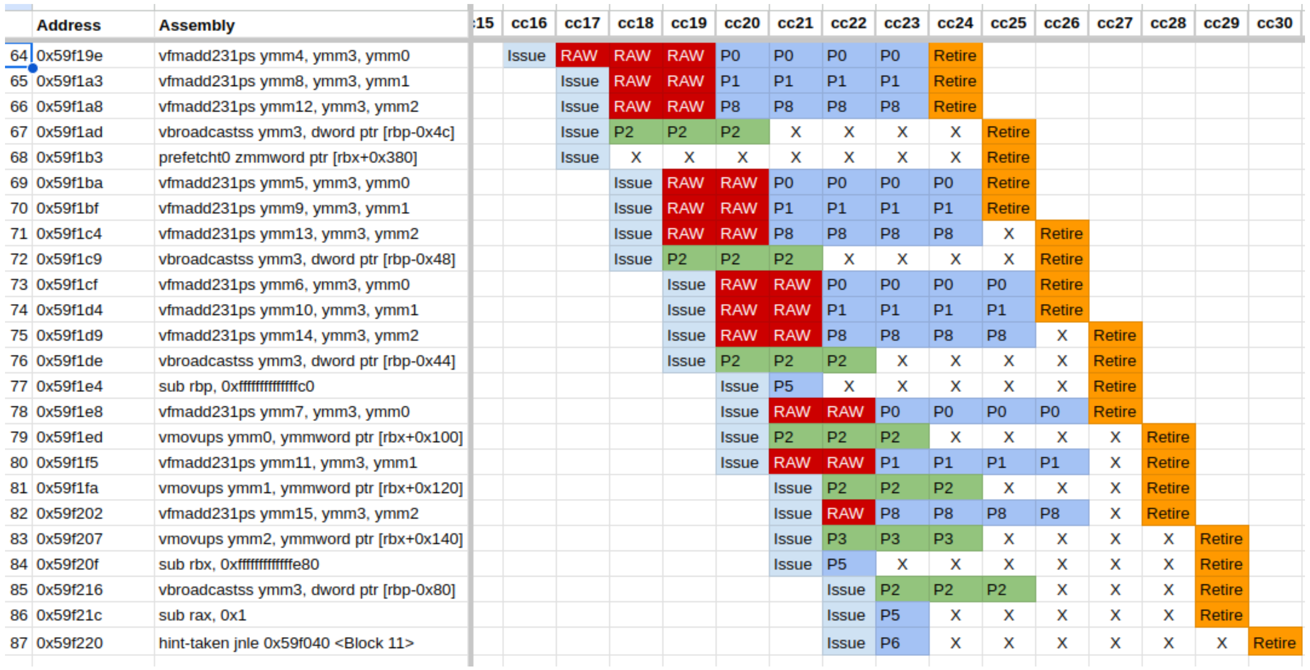
	Address	Assembly	Clock Cycles->	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9	cc10	cc11	cc12	cc13	cc14	cc15	cc16
	0x59f040	**Block 11:**																	
1	0x59f040	vfmadd231ps ymm4, ymm3, ymm0		Issue	P0	P0	P0	P0	Retire										
2	0x59f045	vfmadd231ps ymm8, ymm3, ymm1		Issue	P1	P1	P1	P1	Retire										
3	0x59f04a	vfmadd231ps ymm12, ymm3, ymm2		Issue	P8	P8	P8	P8	Retire										
4	0x59f04f	prefetcht0 zmmword ptr [rbp+0x100]		Issue	X	X	X	X	Retire										
5	0x59f056	vbroadcastss ymm3, dword ptr [rbp-0x7c]			Issue	P2	P2	P2	X	Retire									
6	0x59f05c	prefetcht0 zmmword ptr [rbx+0x240]			Issue	X	X	X	X	Retire									
7	0x59f063	vfmadd231ps ymm5, ymm3, ymm0			Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire						
8	0x59f068	vfmadd231ps ymm9, ymm3, ymm1			Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire						
9	0x59f06d	vfmadd231ps ymm13, ymm3, ymm2				Issue	RAW	RAW	P8	P8	P8	P8	Retire						
10	0x59f072	vbroadcastss ymm3, dword ptr [rbp-0x78]				Issue	P2	P2	P2	X	X	X	Retire						
11	0x59f078	vfmadd231ps ymm6, ymm3, ymm0				Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire					
12	0x59f07d	vfmadd231ps ymm10, ymm3, ymm1				Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire					
13	0x59f082	vfmadd231ps ymm14, ymm3, ymm2					Issue	RAW	RAW	P8	P8	P8	P8	Retire					
14	0x59f087	vbroadcastss ymm3, dword ptr [rbp-0x74]					Issue	P2	P2	P2	X	X	X	Retire					
15	0x59f08d	prefetcht0 zmmword ptr [rbx+0x280]					Issue	X	X	X	X	X	X	X	Retire				
16	0x59f094	vfmadd231ps ymm7, ymm3, ymm0					Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire				
17	0x59f099	vmovups ymm0, ymmword ptr [rbx-0x20]						Issue	P3	P3	P3	X	X	X	Retire				
18	0x59f09e	vfmadd231ps ymm11, ymm3, ymm1						Issue	RAW	RAW	P1	P1	P1	P1	Retire				
19	0x59f0a3	vmovups ymm1, ymmword ptr [rbx]						Issue	P2	P2	P2	X	X	X	X	Retire			
20	0x59f0a7	vfmadd231ps ymm15, ymm3, ymm2						Issue	RAW	RAW	P8	P8	P8	P8	X	Retire			
21	0x59f0ac	vmovups ymm2, ymmword ptr [rbx+0x20]							Issue	P3	P3	P3	X	X	X	Retire			
22	0x59f0b1	vbroadcastss ymm3, dword ptr [rbp-0x70]							Issue	P2	P2	P2	X	X	X	Retire			
23	0x59f0b7	vfmadd231ps ymm4, ymm3, ymm0							Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire		
24	0x59f0bc	vfmadd231ps ymm8, ymm3, ymm1							Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire		
25	0x59f0c1	vfmadd231ps ymm12, ymm3, ymm2								Issue	RAW	RAW	P8	P8	P8	P8	Retire		
26	0x59f0c6	vbroadcastss ymm3, dword ptr [rbp-0x6c]								Issue	P2	P2	P2	X	X	X	Retire		
27	0x59f0cc	vfmadd231ps ymm5, ymm3, ymm0								Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire	
28	0x59f0d1	vfmadd231ps ymm9, ymm3, ymm1								Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire	
29	0x59f0d6	vfmadd231ps ymm13, ymm3, ymm2									Issue	RAW	RAW	P8	P8	P8	P8	Retire	
30	0x59f0db	vbroadcastss ymm3, dword ptr [rbp-0x68]									Issue	P2	P2	P2	X	X	X	Retire	
31	0x59f0e1	prefetcht0 zmmword ptr [rbx+0x2c0]									Issue	X	X	X	X	X	X	X	Retire

Figure 6.16: Gantt chart of the hot loop instructions with improvement 1



	Address	Assembly	cc8	cc9	cc10	cc11	cc12	cc13	cc14	cc15	cc16	cc17	cc18	cc19	cc20	cc21	cc22	cc23	cc24
32	0x59f0e8	vfmadd231ps ymm6, ymm3, ymm0	Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire								
33	0x59f0ed	vfmadd231ps ymm10, ymm3, ymm1		Issue	RAW	RAW	P1	P1	P1	P1	Retire								
34	0x59f0f2	vfmadd231ps ymm14, ymm3, ymm2		Issue	RAW	RAW	P8	P8	P8	P8	Retire								
35	0x59f0f7	vbroadcastss ymm3, dword ptr [rbp-0x64]		Issue	P2	P2	P2	X	X	X	X	Retire							
36	0x59f0fd	vfmadd231ps ymm7, ymm3, ymm0		Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire							
37	0x59f102	vmovups ymm0, ymmword ptr [rbx+0x40]			Issue	P2	P2	P2	X	X	X	Retire							
38	0x59f107	vfmadd231ps ymm11, ymm3, ymm1			Issue	RAW	RAW	P1	P1	P1	P1	Retire							
39	0x59f10c	vmovups ymm1, ymmword ptr [rbx+0x60]			Issue	P3	P3	P3	X	X	X	X	Retire						
40	0x59f111	vfmadd231ps ymm15, ymm3, ymm2			Issue	RAW	RAW	P8	P8	P8	P8	X	Retire						
41	0x59f116	vmovups ymm2, ymmword ptr [rbx+0x80]				Issue	P2	P2	P2	X	X	X	Retire						
42	0x59f11e	vbroadcastss ymm3, dword ptr [rbp-0x60]				Issue	P3	P3	P3	X	X	X	Retire						
43	0x59f124	vfmadd231ps ymm4, ymm3, ymm0				Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire					
44	0x59f129	vfmadd231ps ymm8, ymm3, ymm1				Issue	RAW	RAW	RAW	P1	P1	P1	P1	Retire					
45	0x59f12e	prefetcht0 zmmword ptr [rbx+0x300]					Issue	X	X	X	X	X	X	Retire					
46	0x59f135	vfmadd231ps ymm12, ymm3, ymm2					Issue	RAW	RAW	P8	P8	P8	P8	Retire					
47	0x59f13a	vbroadcastss ymm3, dword ptr [rbp-0x5c]					Issue	P2	P2	P2	X	X	X	X	Retire				
48	0x59f140	vfmadd231ps ymm5, ymm3, ymm0					Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire				
49	0x59f145	vfmadd231ps ymm9, ymm3, ymm1						Issue	RAW	RAW	P1	P1	P1	P1	Retire				
50	0x59f14a	vfmadd231ps ymm13, ymm3, ymm2						Issue	RAW	RAW	P8	P8	P8	P8	Retire				
51	0x59f14f	vbroadcastss ymm3, dword ptr [rbp-0x58]						Issue	P2	P2	P2	X	X	X	X	Retire			
52	0x59f155	vfmadd231ps ymm6, ymm3, ymm0						Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire			
53	0x59f15a	vfmadd231ps ymm10, ymm3, ymm1							Issue	RAW	RAW	P1	P1	P1	P1	Retire			
54	0x59f15f	vfmadd231ps ymm14, ymm3, ymm2							Issue	RAW	RAW	P8	P8	P8	P8	Retire			
55	0x59f164	vbroadcastss ymm3, dword ptr [rbp-0x54]							Issue	P2	P2	P2	X	X	X	X	Retire		
56	0x59f16a	prefetcht0 zmmword ptr [rbx+0x340]							Issue	X	X	X	X	X	X	X	Retire		
57	0x59f171	vfmadd231ps ymm7, ymm3, ymm0								Issue	RAW	RAW	P0	P0	P0	P0	Retire		
58	0x59f176	vmovups ymm0, ymmword ptr [rbx+0xa0]								Issue	P2	P2	P2	X	X	X	Retire		
59	0x59f17e	vfmadd231ps ymm11, ymm3, ymm1								Issue	RAW	RAW	P1	P1	P1	P1	X	Retire	
60	0x59f183	vmovups ymm1, ymmword ptr [rbx+0xc0]								Issue	P3	P3	P3	X	X	X	X	Retire	
61	0x59f18b	vfmadd231ps ymm15, ymm3, ymm2									Issue	RAW	P8	P8	P8	P8	X	Retire	
62	0x59f190	vmovups ymm2, ymmword ptr [rbx+0xe0]									Issue	P2	P2	P2	X	X	X	Retire	
63	0x59f198	vbroadcastss ymm3, dword ptr [rbp-0x50]									Issue	P3	P3	P3	X	X	X	X	Retire

Figure 6.17: Gantt chart of the hot loop instructions with improvement 2

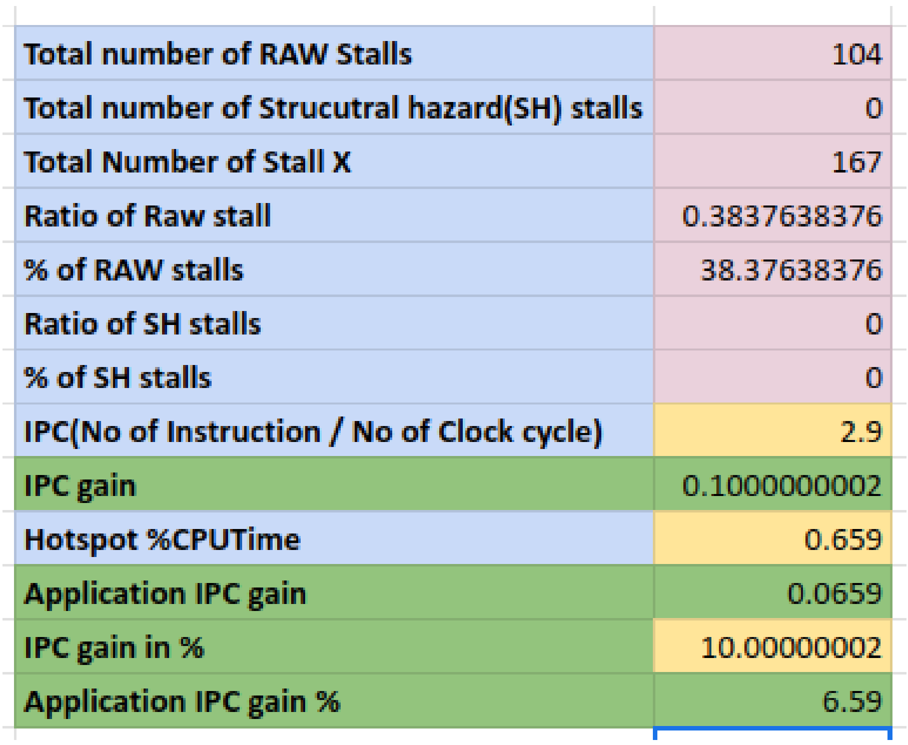


	Address	Assembly	:15	cc16	cc17	cc18	cc19	cc20	cc21	cc22	cc23	cc24	cc25	cc26	cc27	cc28	cc29	cc30
64	0x59f19e	vfmadd231ps ymm4, ymm3, ymm0		Issue	RAW	RAW	RAW	P0	P0	P0	P0	Retire						
65	0x59f1a3	vfmadd231ps ymm8, ymm3, ymm1			Issue	RAW	RAW	P1	P1	P1	P1	Retire						
66	0x59f1a8	vfmadd231ps ymm12, ymm3, ymm2			Issue	RAW	RAW	P8	P8	P8	P8	Retire						
67	0x59f1ad	vbroadcastss ymm3, dword ptr [rbp-0x4c]			Issue	P2	P2	P2	X	X	X	X	Retire					
68	0x59f1b3	prefetcht0 zmmword ptr [rbx+0x380]			Issue	X	X	X	X	X	X	X	Retire					
69	0x59f1ba	vfmadd231ps ymm5, ymm3, ymm0				Issue	RAW	RAW	P0	P0	P0	P0	Retire					
70	0x59f1bf	vfmadd231ps ymm9, ymm3, ymm1				Issue	RAW	RAW	P1	P1	P1	P1	Retire					
71	0x59f1c4	vfmadd231ps ymm13, ymm3, ymm2				Issue	RAW	RAW	P8	P8	P8	P8	X	Retire				
72	0x59f1c9	vbroadcastss ymm3, dword ptr [rbp-0x48]				Issue	P2	P2	P2	X	X	X	X	Retire				
73	0x59f1cf	vfmadd231ps ymm6, ymm3, ymm0					Issue	RAW	RAW	P0	P0	P0	P0	Retire				
74	0x59f1d4	vfmadd231ps ymm10, ymm3, ymm1					Issue	RAW	RAW	P1	P1	P1	P1	Retire				
75	0x59f1d9	vfmadd231ps ymm14, ymm3, ymm2					Issue	RAW	RAW	P8	P8	P8	P8	X	Retire			
76	0x59f1de	vbroadcastss ymm3, dword ptr [rbp-0x44]					Issue	P2	P2	P2	X	X	X	X	Retire			
77	0x59f1e4	sub rbp, 0xffffffffffffffc0						Issue	P5	X	X	X	X	X	Retire			
78	0x59f1e8	vfmadd231ps ymm7, ymm3, ymm0						Issue	RAW	RAW	P0	P0	P0	P0	Retire			
79	0x59f1ed	vmovups ymm0, ymmword ptr [rbx+0x100]						Issue	P2	P2	P2	X	X	X	X	Retire		
80	0x59f1f5	vfmadd231ps ymm11, ymm3, ymm1						Issue	RAW	RAW	P1	P1	P1	P1	X	Retire		
81	0x59f1fa	vmovups ymm1, ymmword ptr [rbx+0x120]							Issue	P2	P2	P2	X	X	X	Retire		
82	0x59f202	vfmadd231ps ymm15, ymm3, ymm2							Issue	RAW	P8	P8	P8	P8	X	Retire		
83	0x59f207	vmovups ymm2, ymmword ptr [rbx+0x140]							Issue	P3	P3	P3	X	X	X	X	Retire	
84	0x59f20f	sub rbx, 0xfffffffffffffe80							Issue	P5	X	X	X	X	X	X	Retire	
85	0x59f216	vbroadcastss ymm3, dword ptr [rbp-0x80]								Issue	P2	P2	P2	X	X	X	Retire	
86	0x59f21c	sub rax, 0x1								Issue	P5	X	X	X	X	X	Retire	
87	0x59f220	hint-taken jnle 0x59f040 <Block 11>								Issue	P6	X	X	X	X	X	X	Retire

Figure 6.18: Gantt chart of the hot loop instructions with improvement 3

Thus, we will consider an additional FP FMA execution unit at port no. 8. Then we again see the theoretical pipeline view of the hot loop of our hotspot function, see Figure. 6.16, 6.17, 6.18.

Now the result of this new pipeline view is given in the next Figure. 6.19.



Total number of RAW Stalls	104
Total number of Strucutral hazard(SH) stalls	0
Total Number of Stall X	167
Ratio of Raw stall	0.3837638376
% of RAW stalls	38.37638376
Ratio of SH stalls	0
% of SH stalls	0
IPC(No of Instruction / No of Clock cycle)	2.9
IPC gain	0.1000000002
Hotspot %CPUTime	0.659
Application IPC gain	0.0659
IPC gain in %	10.00000002
Application IPC gain %	6.59

Figure 6.19: Summary of the above Gantt Chart with new execution unit

Now, to verify whether the extra execution port is giving us improvement, we found out the instructions per cycle (IPC) for both pipeline views and then found out the IPC gain in the second case, Both shown in the Figure.6.15,6.19.

From both the summaries, we see that by adding an extra floating point fused multiply add unit for the inference of BERT Base Uncased workload, we got an IPC gain of **10%** for the top hotloop of hotspot function and Overall IPC gain of **6.59%** for the hotspot function.

Chapter 7

Conclusion and Future Work

The study and experiments in this project help us understand how deep learning models work on a CPU, not just from a software point of view but also how they interact with the CPU's internal design. It shows how the model and the hardware work together during inference.

Based on our findings, here are some possible next steps:

- since multithreading was disabled for our experiments, we shall enable them and repeat the same microarchitecture analysis, and confirm if the results that we got holds.
- A similar methodology can be applied to RISC-V based architecture such as the SiFive 650, which are well-suited for AI and machine learning workloads.
- Explore how compilation strategies (e.g., MKL-DNN optimizations, operator fusion, quantization) and custom kernels affect microarchitectural usage and inference performance.

The project performs a methodical and comprehensive analysis of the inference workload for BERT-base uncased model. The study initially started with understanding deep-learning and its frameworks. Then we went on to check what are the profiling tools and what are the methodology we need to focus on profiling our model.

Next we understood the microarchitecture of the processor so that we will be able to understand how the skylake microarchitecture works. Then we setup the environment to carry out our experiments, starting from downloading the required libraries for our model and VTune profiler for profiling.

Finally, the project culminates in a microarchitectural exploration and hotspot analysis that identifies key performance bottlenecks. Based on these insights, a theoretical solution is proposed—introducing an additional execution port—to mitigate the bottleneck and improve overall performance.

Bibliography

[1] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844459

[2] IBM, "Deep learning - ibm think," https://www.ibm.com/think/topics/deep-learning, 2024.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint*, vol. arXiv:1511.08458, 2015. [Online]. Available: https://arxiv.org/abs/1511.08458

[4] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint*, vol. arXiv:1912.05911, 2019. [Online]. Available: https://arxiv.org/abs/1912.05911

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Łukasz Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint*, vol. arXiv:1706.03762, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[6] L. PyTorch, "Learn pytorch - tutorials and resources," https://www.learnpytorch.io/, 2024.

[7] DeepLearning.AI, "Natural language processing," 2024, accessed: 2025-04-04. [Online]. Available: https://www.deeplearning.ai/resources/natural-language-processing/

[8] J. Alammar, "The illustrated transformer," https://jalammar.github.io/illustrated-transformer/, 2018.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint*, vol. arXiv:1810.04805, 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[10] I. Corporation, "Top-down microarchitecture analysis method," https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html, 2023.

[11] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 11th ed. Pearson Education, 2020. [Online]. Available: https://www.pearson.com/en-us/subject-catalog/p/computer-organization-and-architecture/P200000003394/9780135205129

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012. [Online]. Available: https://acs.pub.ro/~cpop/SMPA/Computer%20Architecture%20A%20Quantitative%20Approach%20(5th%20edition).pdf

[13] I. Corporation, "Element-wise primitive (eltwise)," 2023, accessed: 2025-04-09. [Online]. Available: https://www.intel.com/content/www/us/en/docs/onednn/developer-guide-reference/2023-2/eltwise-001.html

[14] A. Fog, *Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2024, last accessed: April 2025. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[15] H. Face, "Transformers documentation," https://huggingface.co/docs/transformers/en/index, 2024.

[16] ——, “Bert-base uncased model,” https://huggingface.co/google-bert/bert-base-uncased, 2024.

[17] DeepLearning.AI, “Deep learning resources,” https://www.deeplearning.ai/, 2024.

[18] Wikipedia contributors, “Skylake (microarchitecture) — wikipedia, the free encyclopedia,” 2024, accessed: 2025-04-10. [Online]. Available: https://en.wikipedia.org/wiki/Skylake_(microarchitecture)

[19] Intel Corporation, “Developer documentation,” https://www.intel.com/content/www/us/en/resources-documentation/developer.html, 2024, accessed: 2025-04-10.