POLITECNICO DI TORINO

Master's Degree in Degree in Computer Engineering

Master's Degree Thesis

Inference optimization of Large Language Models on RISC-V HPC platforms

Supervisors

Prof. Daniele Jahier PAGLIARI

Dr. Alessio BURRELLO

Phd. Mohamed Amine HAMDI

Phd. Cyril KOENIG

Prof. Luca BENINI

Candidate

Javier Jesus POVEDA RODRIGO

October 2024

Abstract

Over the past decade, there have been significant improvements in Artificial Intelligence (AI), more particularly in the area of natural language processing (NLP) thanks to the emergence of Large Language Models (LLMs). These models are empowering many deep learning applications such as translation, text and image generation and many others. These transformer-based models bring to the table new challenges from a computational point of view, due to its characteristic attention mechanism and embeddings.

Even though these types of workloads are typically offloaded to GPUs, there are several applications and use cases that require CPU as the workhorse because of its reduced cost and bigger flexibility and expandability in memory terms. These increasingly relevant CPU-based solutions have been developed in more classical ISAs such as x86 and ARM and extensive research is being conducted to enhance AI applications on these platforms, aiming to fully unlock the already available high computational power. Recently, RISC-V many-core SoCs are arising as an open-source alternative. Despite being capable of performing High Performance Computing (HPC) workloads, lack reliable support in their toolchains and libraries, not properly targeting the different Hardware platforms or not displaying the level of optimization compared to x86/ARM counterparts.

For instance, Basic Linear Algebra Subprograms (BLAS) libraries, core in AI-centric workloads, are not optimal for RISC-V and lack support for different Vector extension versions. Most of the available alternatives such as auto-vectorization are not reliable enough or not supported for vector enabled RISC-V cores.

Thus, the main target of this thesis is to analyze and address the current status of the toolchains and the critical bottlenecks in LLMs inference, trying to improve the performance of inference with LLMs with the state-of-the-art models and frameworks on RISC-V multi-core CPUs, which is increasingly gathering interest for its potential for HPC applications.

This thesis builds upon the llama.cpp open-source inference framework and its ggml tensor library backend. Our work relies on performing several many-core-aware modifications and optimizations on top of LLAMA.cpp, such as NUMA-aware thread dispatch and tuning of thread spawning based on the computation.

Additionally, we propose a new GEMM and GEMV implementations, able to exploit vector extensions and other basic optimization techniques (e.g. loop unrolling and weight sharing), through which we avoid incurring substantial overheads and are able to achieve improved performances.

We carried out experiments on the MILK-V PIONEER, a system based around the RISC-V CPU Sophon SG2042. The SG2042 chip features a Network-on-Chip (NoC) architecture with 64 T-Head C920 cores distributed in 16 clusters with 4 NUMA memory regions and three levels of cache memory. Each core is clocked up to 2GHz and is built around the Base Instruction set IMAFDC, with a 9 to 12 stages pipeline with multiple advanced functionalities, including the support of the RISC-V vector instruction extension on its draft version 0.7.1. In summary, this work leverages several strategies to improve inference, exploring its multi-dimensional nature for optimization and performance bottlenecks. It aims to demonstrate the potential of RISC-V systems in modern HPC workloads, particularly in LLM inference tasks.

Acknowledgements

First of all, I must thank my tutors, Daniele and Alessio. You trusted me, provided key guidance, and gave me the opportunity to learn and work on such a complex and cutting-edge research project. Huge thanks to Amine and all the dedication, advice, and help that you provided; without it, this project would not have been possible at all. All three of you showed me how exceptional you are, not only with your technical expertise but also with your passion and care for your team and colleagues. Also, thanks to the whole Lab 4, I could not have felt more comfortable in such a great working atmosphere and an incredible group of people.

I would like to especially mention my parents and sister, who supported me in these two hard years, during which distance was always felt. Without their help and aid, I would have never been capable of doing this journey.

A big thanks to my all time friends, that made the distance feel a bit smaller with our calls and always welcomed me with excitement, gossip and some beers.

Also, I can't forget of all the friends, master colleagues and people that I have met during this two-year experience in Southampton and Torino. You all made this experience completely life-changing and I know I will keep good friends for many years to come, with many anecdotes and share experience to laugh about.

Finally, I would like to dedicate these words to the person who has been my partner during this journey and had to put up with me during the whole process of this thesis. Amore, this is just another opportunity to be grateful to have you in my life, being my light and bringing fun and smiles to my day-to-day. I will never be able to express how lucky I feel to have met you (on a random day, in a random town, in a random country) and that you are now a central part of my life. Also, I will not be able to thank you enough for all the support and patience you gave me during this harsh period, being my safe space in the hardest moments and loving me despite feeling unloveable. Muchas gracias por ser una bellísima persona y ser como tú eres, no podría haber elegido una mejor pareja para caminar en este largo

y complicado camino, y lo mucho que nos queda por delante. La distancia no ha sido nuestra mejor amiga, pero si tengo que elegir a alguien con quien resistirla, lo haría contigo, una y mil veces, esperándote con ilusión y cariño en la estación de tren, y dejándote entre lágrimas al despedirnos. Grazie di cuore per tutto, T'estime moltíssim mi vida <3

"Cuando quieres algo, todo el Universo conspira para que realices tu deseo."
El Alquimista, Paulo Coelho

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

BPTT

Backpropagation Through Time

BRNN

Bidirectional Recurrent Neural Network

BLAS

Basic Linear Algebra Subprograms

CPU

Central Processing Unit

CRS

Control and Status Register

DNN

Deep Neural Network

FNN

Feedfoward Neural Network

FPU

Floating-Point Unit

GRU

Gated Recurrent Unit

GPT	Generative Pre-trained Transformer
GEMM	General Matrix Multlication
HPC	High Performance Computing
HBM	High Bandwidth Memory
ISA	Instruction Set Architecture
LLM	Large Language model
LSTM	Long Short-Term Memory Unit
MLP	Multi-Layer Perceptron
NLP	Natural Language Processing
NN	Neural Network
NLM	Neural Language Model
NUMA	Non-Uniform Memory Access
NoC	Network-on-Chip

RISC

Reduced Instruction Set Computing

RISC-V

Reduced Instruction Set Computing on its fifth version

RNN

Recurrent Neural Network

SLM

Statistical Language Model

TLB

Translation Lookaside Buffer

Chapter 1

Introduction

Artificial intelligence has become a transformative element of contemporary society, driven by unprecedented advancements in the area of NLP (Natural Language Processing). At the forefront of this revolution are the LLMs (Large Language models), such as ChatGPT, Claude or Gemini, which showcased remarkable capabilities in tasks such as complex language understanding and text generation. Recent advances integrate multi-modality, evolving from pure text to audio and image integrated into a single model. These exceptional results of LLMs (Large Language models) have led to an explosion in the area of artificial intelligence, being able to integrate in many industries, utilities, and daily life.

In parallel, the landscape of HPC (High Performance Computing) has rapidly grown due to the increasing demands of scientific simulation, data analytics and even the recent development of these gigantic language models. In this sphere of computing, x86 architectures became the dominant ones at the hands of multinational companies such as Intel and AMD for decades. Despite the maturity of x86, other architectures have gained significant traction to compete in this area, mainly ARM and RISC-V. The excellent results of ARM-based supercomputers, like the Fugaku in Japan, have highlighted the viability also for RISC architectures to achieve exascale performance while improving power profiles.

In this context, RISC-V crops up as an attractive alternative, with its open-source ISA (Instruction Set Architecture) and flexibility to incorporate custom extensions and specialised instructions tailored for these highly demanding computer tasks. Despite its maturity in other areas related to simpler architectures like microcontrollers and custom ASICs, is continuously improved to become a crucial node also in more complex systems such as CPUs. Many efforts are targeting this archiecture as the European Processor Initiative and hardware developments in multiple research institutions and companies aim to escalate this technology and make this open-source ISA (Instruction Set Architecture) a competitive alternative.

Therefore, due to its fresh perspective and high potential for fine-grained optimization, RISC-V has positioned as a promising platform for tackling the challenges posed by AI workloads in HPC environments.

This thesis addresses the challenge of adapting current inference frameworks in order to exploit the full functionality of state-of-the-art models with HPC capable RISC-V platforms. Therefore, this piece of main contrivutions are as follows:

- Exploration of hardware architecture and capabilities with current best practices on multi-core RISC-V CPUs and the current software ecosystem.
- Custom implementation of llama.cpp open source inference framework with inclusion of novel kernel-level optimisations and higher-level improvements in execution such as NUMA awareness.

All of these are carried out in practice, with real hardware execution and a multidimensional approach that tackles the several problems or improvement points that current setups utilize for this not-so-explored usage of RISC-V when used as an HPC platform. The thesis is organized in the following sections: Chapter 2 provides background context on the developments of NLP and AI, the basics of RISC-V ISA and the main components related with the thesis and HPC. Chapter 3 describes the hardware used as RISC-V HPC platform and introduces specific tools and software modules used during the execution of this thesis. Chapter 4 gives an overview of the different methodologies used and the logical steps taken, considering the relevant new information also obtained, through the different sections and difficulties endured. Chapter 5 focuses on visualizing and analyzing the most significant results achieved during this project, describing in an organized manner from the most simple ones in exploitation and single core to multi-threaded operation. Chapter 6 is dedicated to providing concluding remarks and discussing future work.

Chapter 2

Background

2.1 NLP and AI

Language is the foundation of knowledge storage and knowledge exchange, being a key element in humanity. Due to its huge importance, many efforts have been directed towards providing machines with the capability of comprehending and interacting with natural language seamlessly as humans. The first efforts to achieve this goal can be traced back to the early beginnings in the 1990s.

2.1.1 History of ML for NLP

The first attempts to tackle this challenge took a probabilistic statistical perspective with the SLMs (Statistical Language Models) [1]. However, these early approaches were hampered by their limited context due to storage limitations, as each word could only relate to its first one or two preceding words, resulting in a reduction in accuracy.

The rapid development of NNs (Neural Networks) and deep learning during the 2000s made these statistical models evolve. This new paradigm brought new tools such as the first versions of word embeddings techniques, which represented words as fixed-dimensional vectors. These dimensions and the angles between words aimed to capture the semantic relationships between them. This behavior can be showcased with the example of the very first widely spread tool at the time, Word2Vec [2], where $vector(''King'') - vector(''Man'') + vector(''Woman'')$ results in the vector equivalent to "Queen". This NLMs (Neural Language Models) also integrated a key architectural development of neural networks, activation functions. Inside of the hidden layers of the models, which recieve as input the word vectors, they used these activation functions, typically a sigmoid or a tanh, to resemble the behavior of biological neurons. After the different hidden layers operations, the final vector undergoes the Softmax function, producing an output

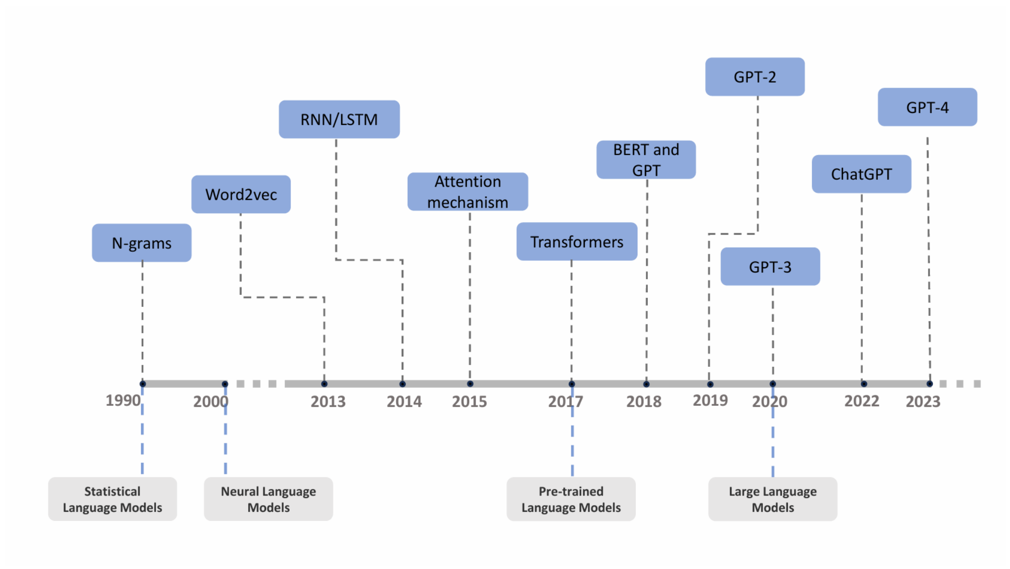


Figure 2.1: History and relevant developments in NLP and languaje models. Extracted from [1].

vector containing the probability distribution assigned to each word over the entire vocabulary of the model. Furthermore, new neural network architectures appeared and became the de facto standard for the field, more in particular RNN (Recurrent Neural Network). These architectures enable to extend the functionality of Feedfoward Neural Networks (or MLPs (Multi-Layer Perceptrons)) by adding a time dimensionality with a feedback element (as shown in Fig. 2.2), being able to take into account previous inputs in the execution flow [3].

Although this new approach brought better performances it suffered from a crucial problem, the vanishing or exploding gradient. This causes the contribution of states that took place before that the current time steps can tend to stop (vanishing) or can weigh too much and provoke heavy changes (exploding) [4]. This led to the inclusion of the BPTT (Backpropagation Through Time), which adapts the feedback algorithm for RNNs, unfolding them constructing traditional Feedfoward Neural Networks as shown in figure 2.3.

Mainly two new improvements were included to deal with these limitations.The LSTMs (Long Short-Term Memory Units) firstly introduced in 1997 [6] and its lighter version the GRU (Gated Recurrent Unit) brought up the concept of memory cells, units of computation that replace the traditional nodes in the hidden layers of a network. Its main function is to help the network determine whether the inputs should go into a memory state or not and if the content of the memory state should have contributed to the output of the model [7]. The other big architectural improvement also comes from a paper of 1997 [8] with the BRNNs (Bidirectional

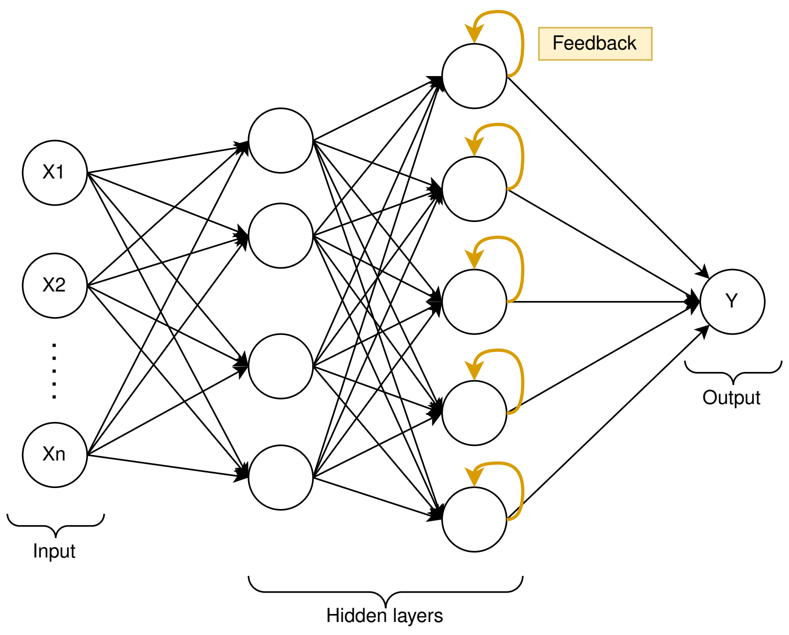


Figure 2.2: RNN (Recurrent Neural Network) simplified diagram with the differential feedback (yellow) compared to standard Feedfoward Neural Networks.

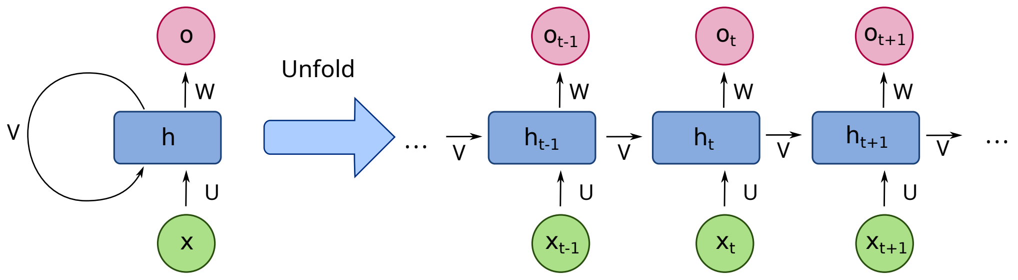


Figure 2.3: Diagram for a one-unit RNN (left) to unfold version (right). From bottom to top: input state (x), hidden state (h), output state (o). U, V, W are the weights of the network.Extracted from [5].

Recurrent Neural Networks), in which the information from the preceding time steps (past) and the subsequent time steps (future) can be used to influence the output at any point of the input sequence, contrasting the previous approaches that only considered the past input as relevant for the output result. Both of these proposals were even combined to have to outperform the current state of the art in different tasks.

In 2014, the NLP field experienced a major breakthrough with the introduction of the sequence-to-sequence (seq2seq) [9] architecture. It used an encoder-decoder architecture, where the encoder processes with different RNNs a variable-length sequence as input generating an intermediate hidden state. Subsequently, it is processed by the decoder, whose task is to work as a conditional language model, using the encoded hidden state as input and the past context to predict the subsequent output [7].

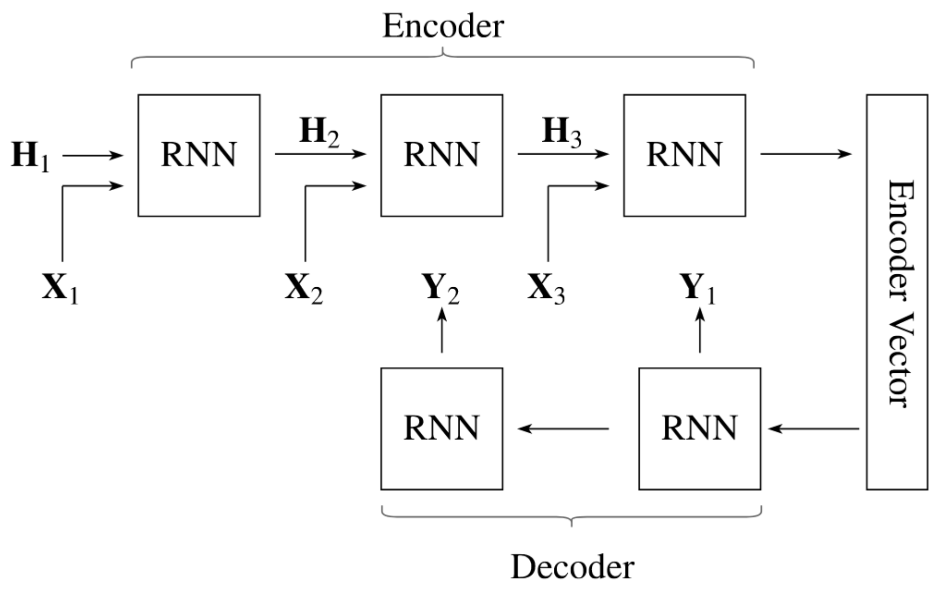


Figure 2.4: Diagram of the Sequence to Sequence (seq2seq) model. From bottom to top: output (Y_t), input (X_t), hidden states from the RNN including past information (H_t). t represents each time step. Extracted from [3].

The main bottleneck of this architecture was the Encoder Vector (also known as the context information), as it encapsulates all the necessary information from the input sequence in a fixed-lenght vector. This constraint proved to be specially problematic for long sequences, resulting in a loss of information and degradation of performace at more complex tasks. Attention Mechanism was introduced to adresss this limitaion.

The Attention Mechanism generally takes two sequences and transforms them into a matrix where each sequence element (such as words or tokens) corresponds to a row or a column. In this particular matrix the values represent the relevant context or correlations between these sub-elements. Two examples of usage of this mechanism are shown in figure 2.5 where in the left image, lighter colors represent a stronger correlation, while in the right one the stronger the opacity of the color, the higher the relationship established by the attention calculation.

This mechanism became the core element of the next most disruptive architecture up to date, the Transformer [11]. This groundbreaking model relies only on

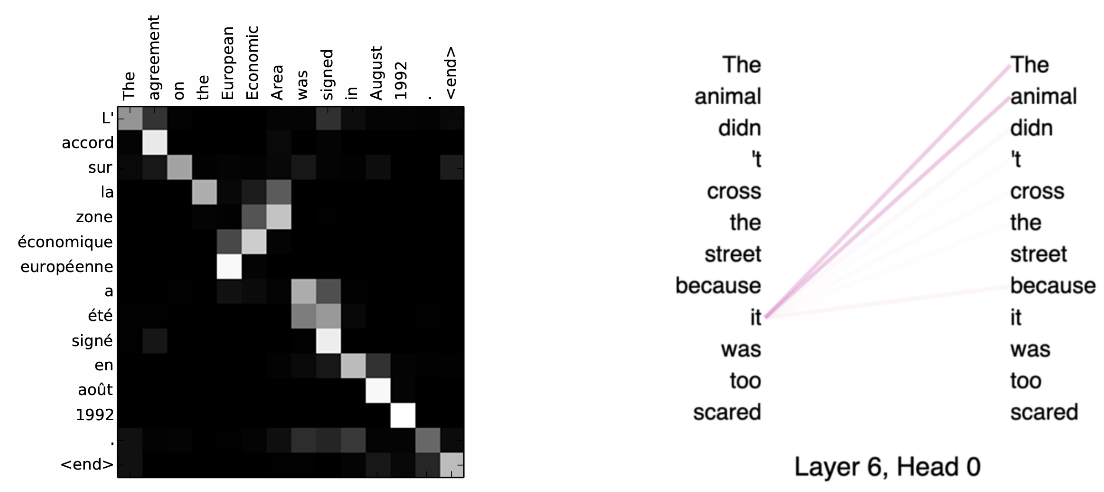


Figure 2.5: Attention Mechanisms examples. Left: : Example of an Alignment matrix of "L'accord sur la zone économique européen a été signé en août 1992" (French) and its English translation "The agreement on the European Economic Area was signed in August 1992"[10].Lighter colors indicate stronger correlations. Right: Self-attention pattern in a GPT-2 model, where color opacity represents the strength of attention relationships

the attention mechanisms and feedforward neural networks, in contrast with the dependency on Recurrent Neural Networks of previous models, utilizing the encoder-decoder architecture and improving the performance by utilizing self-attention for dealing with global or long contexts and dependencies. Its ability to parallelize computation and capabilities with larger contexts led to unprecedented performance improvements, becoming the de facto standard for many NLP tasks. This success turn it the key component of more complex and advanced models, such as the state-of-the-art LLMs (Large Language models).

2.1.2 Transformers and LLMs

First introduced in 2017 in the paper "Attention is all you need" [11], the Transformer introduced several innovative design that included several differences with previous approaches.

The Transformer employs a general encoder-decoder framework, similar to other sequence to sequence models. The encoding structure is composed of several identical encoder modules (e.g. in [11] six encoder components are used). Each module or layer has two sub-layers(or sub-blocks). The first is the attention sub-layer, which performs a multi-headed self-attention operation over the inputs. The second is a classic, position-wise fully connected feed-forward network. The decoding structure is also composed of the same stack of layers but in between it adds an extra attention block, which performs a multi-head attention over the output of the encoder structure. As an input for both structures, it uses learned word embeddings to generate the input fixed-dimension vectors from the tokens/words for the two structures. Additionally, since the model doesn't use recurrence and neither convolution, there is no information about the order of the sequence and thus it is added a "positional encoding" to include this information in the input vectors. A general overview of the described architecture is reflected in Figure 2.6.

The self-attention mechanism is composed of the so-called "Scaled dot-product attention" which uses a query (Q), key (K) and value (V) vectors, the softmax operation and the scaling by the dimensionality of the key and query vectors (d_k). These vectors are the result of projecting the embedding of with the positional encoding input(X) with the different pre-trained weights of the model (W). These operation is described in Equation 2.1 and 2.2 and its architecture in Figure 2.7.

$$Q = X \times W_q, K = X \times W_k, V = X \times W_v \tag{2.1}$$

$$Attention(Q, K, V) = softmax(\frac{(opt)Mask(QK^T)}{\sqrt{d_k}})V \tag{2.2}$$

As can be seen in the equation 2.2, there is an optional masking operation. This represents the difference between the self-attention in the encoding and decoding stacks. While in the encoding stage, all the sub-tokens contribute to each other, in the decoding process, this step is used to prevent positions from using subsequent elements in the calculation, therefore influencing the current output.

As a final element of its revolutionary design is the parallelization thanks to the Multi-headed Attention. As shown at figure 2.7, several heads are working in parallel with the same input (noted as "h" in Figure 2.7 and "N" in Figure 2.6). This allows the model to explore jointly from different sub-spaces at different positions. This not only boosts performance computationally but also improves the overall performance of the model.

LLMs and pre-trained language models build upon this flexible and powerful architecture and are trained on massive text corpora, allowing them to "learn" through its billions of parameters the fundamental language structures such as vocabulary, syntax semantics and logic [1]. Most of these early implementations added different variations to the original Transformer, like only decoder models like Bert [14] or GPT (Generative Pre-trained Transformer)[15] and all of its evolutions by the hand of OpenAI. The complexity has increased dramatically in the last years, together with new advances like Mixture-of-Experts techniques, multimodality and improving in several areas from both performance and capabilities points of view, becoming a key point of today's technology.

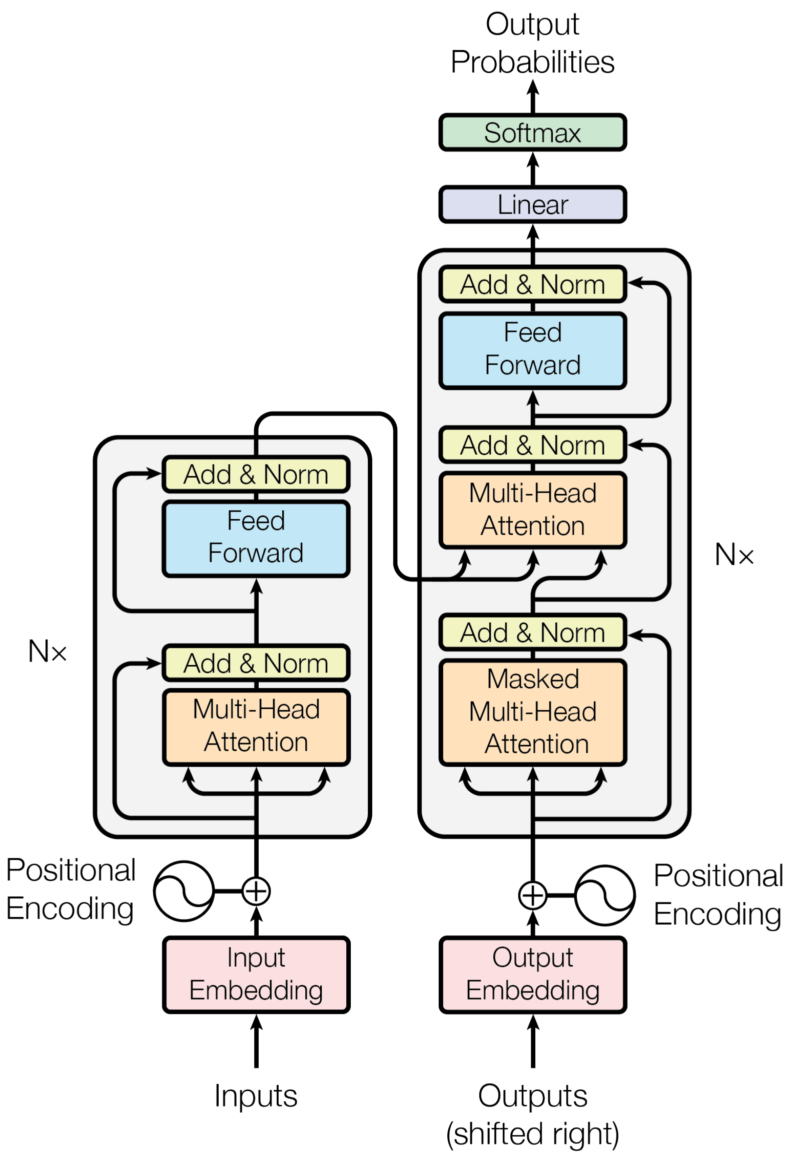


Figure 2.6: Model architecture of the Transformer. Extracted from [11].

2.2 RISC-V ISA

2.2.1 Main characteristics

The landscape computer architecture panorama as been shaped over time by various ISAs (Instruction Set Architectures), each with different characteristics and applications. Dominant proprietary architectures such as x86 or the newer ARM have long histories and extensive ecosystems. However, RISC-V has emerged

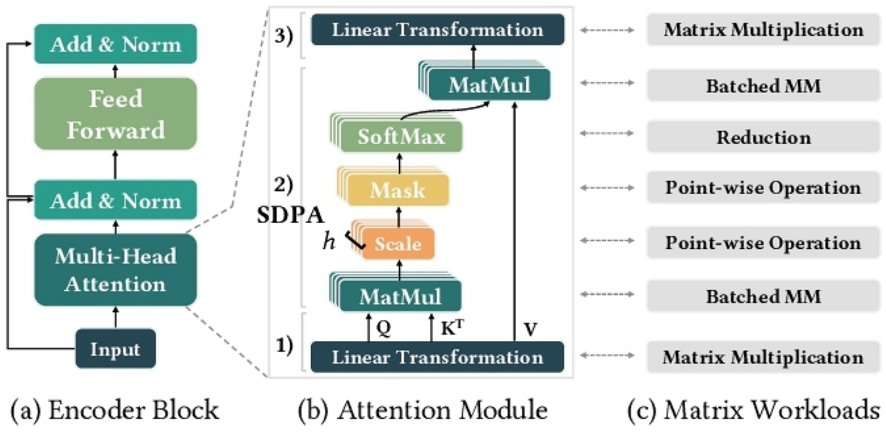


Figure 2.7: The architecture of an encoder block of Scale Dot-Product Attention (SDPA) and the workflow of the attention module. Extracted from [12].

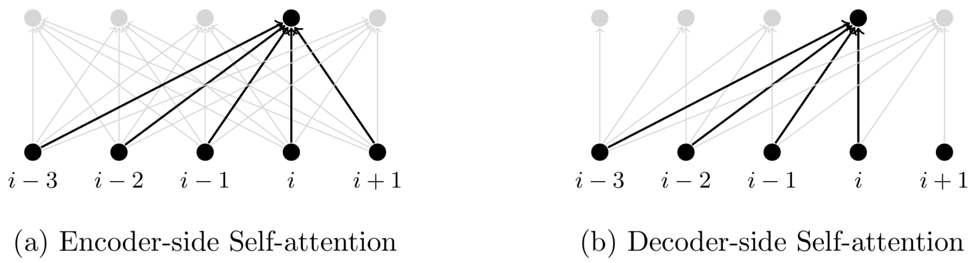

$i-3$
$i-2$
$i-1$
i
$i+1$
(a) Encoder-side Self-attention
$i-3$
$i-2$
$i-1$
i
$i+1$
(b) Decoder-side Self-attention

Figure 2.8: Self-attention on the encoder and decoder sides. Each line connects an input and an output of the self-attention model, indicating a dependency of an output state on an input state. For encoder self-attention, the output at any position is computed by having access to the entire sequence. By contrast, for decoder self-attention,the output at position is computed by seeing only inputs at positions up to i. Extracted from [13].

as the most popular open ISA, grounded in the RISC (Reduced Instruction Set Computing) principles, developing and evolving since its first inception at the University of California, Berkeley in 2010. In contrast to the proprietary ISAs this architecture is completely royalty-free and open-source, establishing a baseline of core instructions that can be further expanded with different extensions [16]. The modularity and extensibility has contributed enormously to its popularity, being able to tailor specific applications and domains. Besides its maturity in the field of embedded processors, in other areas such as high-performance computing or cloud computing it is still under heavy development with newer extensions and improved ecosystem support.

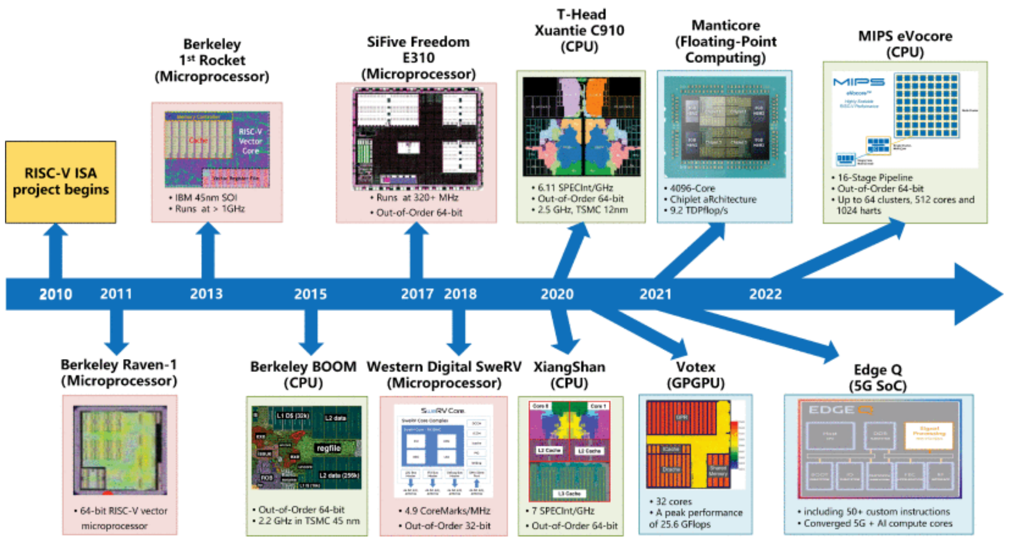


Figure 2.9: Evolution of RISC-V processors. Extracted from [16].

The standardisation and approval of the developments are overseen by the RISC-V International Foundation. This organization is in charge of the key process for taking non-custom extensions, evaluating them, reshaping them through standardization and eventually ratifying them. This whole process is transparent and open to the public, involving several organizations, public (e.g. Linux Foundation) and private in different management roles. This whole process allows for a continued and organized evolution of the RISC-V ISA.

Extensions are classified as either privileged and unprivileged, depending on the usability across all the different privilege levels within the architecture, whose design can vary. This can be subdivided into standard and non-standard, if they are more generic and designed to not conflict with any other preexisting extension or if they are more specialised and can generate conflicts or overlapping with other extensions.

The base functionality includes a minimal set of instructions, with many base specifications for 32-bit/64-bit/128-bit specifications (defined as "RV32I/E, RV64I/E and RV128I"). There are many standard extensions ratified that extend the base functionality such the the "M" the integer multiplication and division or "B" for bit manipulation. When a piece of hardware is compliant with one of this extensions signifies that is incorporates the underlying hardware in order to fully support the instructions related with these extensions. An example of this are arithmetic operations with 16-bit floating point numbers (half-precision). If the CPU or MCU incorporates the "F" extension for single-precision Floating-point but it doesn't the "Zfh" for half-precision extension, inside of a code (e.g C language

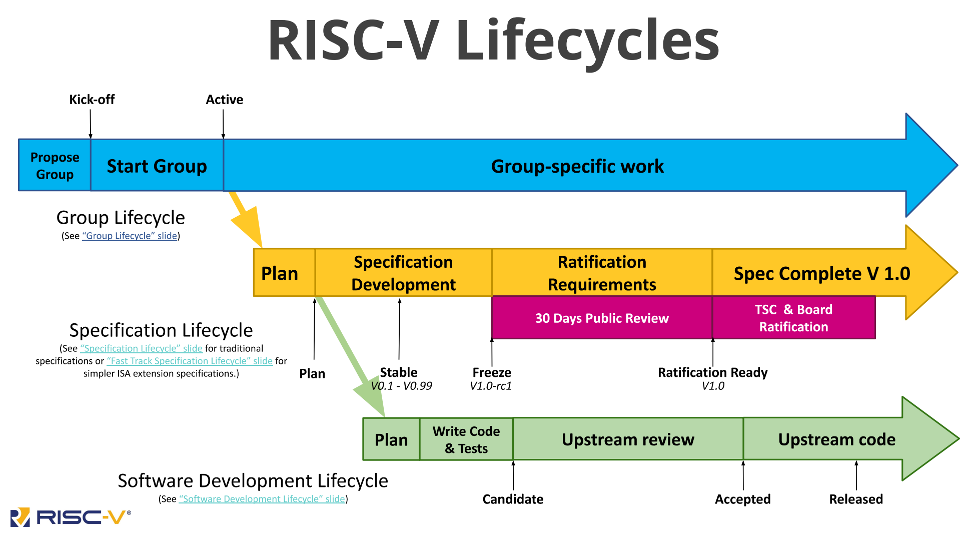


Figure 2.10: Summary diagram of extension lifecycle. Extracted from [17]

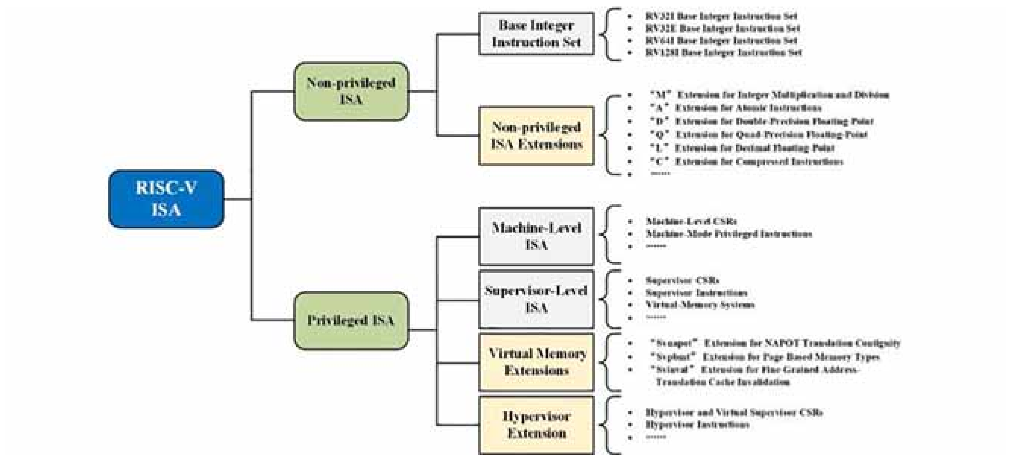

RISC-V ISA
Non-privileged ISA
Privileged ISA
Base Integer Instruction Set
Non-privileged ISA Extensions
Machine-Level ISA
Supervisor-Level ISA
Virtual Memory Extensions
Hypervisor Extension
RV32I Base Integer Instruction Set
RV32E Base Integer Instruction Set
RV64I Base Integer Instruction Set
RV128I Base Integer Instruction Set
"M" Extension for Integer Multiplication and Division
"A" Extension for Atomic Instructions
"D" Extension for Double-Precision Floating-Point
"Q" Extension for Quad-Precision Floating-Point
"L" Extension for Decimal Floating-Point
"C" Extension for Compressed Instructions
......
Machine-Level CSRs
Machine-Mode Privileged Instructions
......
Supervisor CSRs
Supervisor Instructions
Virtual-Memory Systems
......
"Svnapot" Extension for NAPOT Translation Contiguity
"Svpbmt" Extension for Page-Based Memory Types
"Svinval" Extension for Fine-Grained Address-Translation Cache Invalidation
Hypervisor and Virtual Supervisor CSRs
Hypervisor Instructions
......

Figure 2.11: Summary diagram of RISC-V ISA. Extracted from [16].

file) that requires to operate with half-precision numbers supported by the programming language, the compiler (e.g. GCC) will utilize the compliant extensions, in this case, it would promote the numbers and operate in single precision, making more steps than what the direct execution if the extension "Zhf" is supported, and therefore the specific instructions and hardware would be used. All of the currently supported extensions with their versions and specifications can be found in the RISC-V Instruction Set Manual (Volume I and II) [18].

2.2.2 Vector extension

One of the most important extensions that has become stable in the last years is the "V" Vector Operations extension. It is intended to provide support for general data-parallel execution with a small number of instructions compared to other typical packed-SIMD alternatives. It is an open standard for potentially increasing the computational power with the adoption of vector units in RISC-V Cores.

Version evolution: from v0.7 draft to v1.0 stable

The first draft specification the draft spec 0.7 and its first revision for software development 0.7.1 in June of 2019. It aimed to capture the basic required vector functionalities and how some of this will be implemented as vector instructions. This first approch defined a stable enough version in order to settle the bases for developing functional simulators, toolchains and initial implementations of the individual features described. Its realtive stability and generic aproach allowed that some CPUs to be manufactured and desinged implement it, such as the Xuantie C910 (it will be further explained in the following chapters). This settles the basis for the following v0.8, v0.9 and v0.10 preliminary stable releases before the definite v1.0 frozen and ratified version in September of 2021. Despite sharing many features, the hardware supporting version 0.7.1 and the version itself is not directly compatible with the stable version v1.0. The main differences can be summarized int the following list:

- **CRSs (Control and Status Registers) required**: while the 1.0 uses seven unprivileged CRSs (*vstart, vxsat, vxrm, vcsr, vtype, vl, vlenb*), the 0.7 only uses five (*vstart, vxsat, vxrm, vtype, vl*).

- **Vector register grouping**: The extension allows for multiple registers to be grouped together so that with a single vector instruction it can operate on many registers at once. The 0.7 specifications only allowed grouping or a vector length multiplier value (LMUL) greater than 1, in this case, 2, 4, 8. On the other hand, the newer specification also establishes values of LMUL smaller that 1 or fractional, in order to increase the number of effective usable vector register groups when operating on mixed-width values. With the old specification values could only be allocated in a vector register using at least one whole register, however, the newest provides more flexibility and better potential efficiency.

- **Extended instructions and encoding**: over the three intermediate versions more functionalities were extended in order to give as much support and optimized instructions. One of these changes is in the core instruction used for configuring the vector unit, *vsetlv*. This instruction utilizes 3 parameters: the proposed amount of data that is going to be processed for (Application Vector Length) which gets corrected (the specification sets some restrictions) and by establishing the actual number of elements that would be processed *vl* and the *vtype* argument which encodes the size of the individual elements in bits (8,16,32 or 32) and the LMUL value (e.g. for v1.0: 1/8,1/4,1/2,1,2,4,8). The v1.0 specification does not only allow to use of immediate versions (having three in total, compared to the two of the 0.7) of this instruction but also uses 2 extra bits in the instructions encoding "tail" and masking information.

2.3 Compilers

Compilers are key software tools in computer science that are capable of translating high-level programming languages into machine code so it can be executed. With the current paradigm of increasing abstraction and higher-level programs, compilers and their translation task and different optimisations become more and more relevant. Despite being a core element in a language-processing system, it usually doesn't work alone. Before the compiler's execution, the preprocessor takes the source program and generates a modified version for the compiler combining several sources or doing expansions of macros into source language statements. After the Compiler generates the target assembly program, two more actors come into place, the assembler, which produces relocatable machine code and the linker, which, if needed, can put together different relocatable machine codes, resolving any external memory addresses referring to other files. Finally, the loader merges all executable files into memory for execution.

2.3.1 Structure of a compiler

A compiler is composed of two main types of tasks: analysis and synthesis. In the front-end or analysis, the source code is decomposed into tokens and a grammatical structure is imposed. During this process also the correctness of the source code is checked according to the language rules. Another extra step during the analysis process is the collection of information about the source program such as variable names, that are stored in a symbol table. This process consists of four basic steps:

- **Lexical Analysis or scanner**: this first phase analyses the input source code as a stream of characters and groups the sub-strings into meaningful elements called "lexemes". For each of these sub-divisions are output different

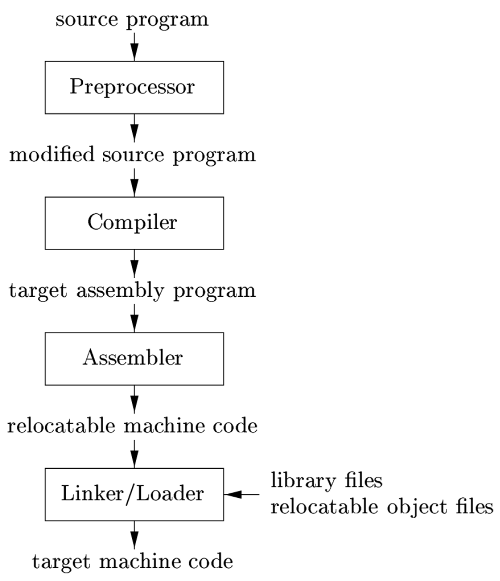


Figure 2.12: Diagram of the different elements and outputs of language-processing systems. Extracted from [19].

tokens. A token can represent operations, like multiplication or assigning, or keywords, but also variables or constants, in which case there is an attribute value also included.

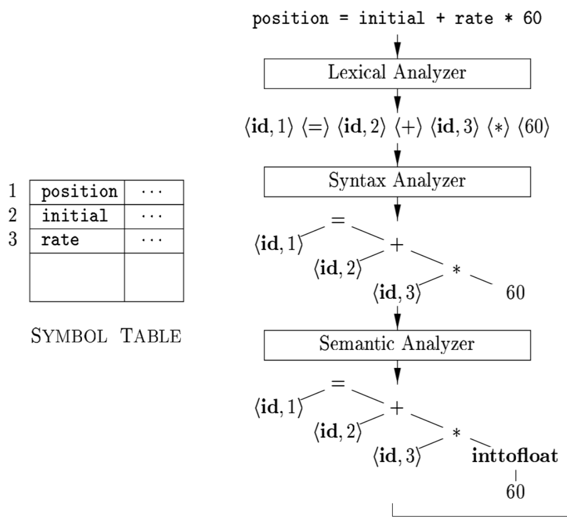
- **Syntax Analysis or parsing**: during this step, a tree-like intermediate representation that represents the grammatical rules is generated. One of the most common is the "syntax tree", where the children of each node represent the arguments of that specific operation and the depth of the order or dependencies between operations (nodes).

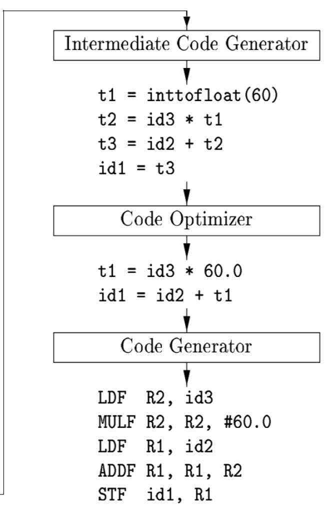
- **Semantic analysis**: through this process, the semantic analyser checks the consistency with the programming language definition of the source code from the syntax tree and the symbol table. One of the key tasks is type checking, where the compiler makes sure that the operands used are correct for the operations(e.g. in C language array indexes must be integers).

- **Intermediate code generation**: many compilers generate a machine-like or low-level (e.g. assembler) intermediate representation that would be fed into the optimization process. These representations are required to be easy to produce and translate into target machine code. One example is the LLVM software, which utilizes a specific Intermediate Representation (IR) created by them.

Following, the intermediate code is processed by the back-end section of the compilers. In this part the code/representation is optimized and tuned for a specific micro-architecture. This whole module encapsulates two main processes:

- **Intermediate Code Optimization**: This is a machine-independent process that seeks to improve the intermediate code. These optimizations can have different targets, such as size or speed.
- **Code Generation**: the software takes the optimized intermediate representation and maps it into the target language. In the case of machine code, the different variables used have to be mapped into the different types of registers or memory, and the specific instructions available by that specific hardware targeted.





```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Figure 2.13: Phases of a compiler with intermediate outputs of translation. Modified version of original extracted from [19].

2.4 Multi-core platforms and parallelissim

During a great period in computer architecture and manufacturing history in the CMOS era, the performance of improvements came from the miniaturization of transistors and the consequent increase in number and the increase in the working speed or clock speeds. Despite this strategy having proven its feasibility increasing at almost an exponential rate the number of transistors and the speed, in the last two decades, newer challenges have encouraged newer designs and architectures.

Problems like the "heat wall" [20] due to power dissipation and power density hinder the advance in this direction and are a challenge for newer generation processors. Another related problem is the slower pace of developments in memory systems such as DRAM (DRAM delay and latency only improved year-over-year a 10% and 20% while the CPU performance since 1986 a 60% [21]). These problems resulted in barriers that potentiated other approaches, like the increase of cores in CPUs, as shown in Figure 2.14.

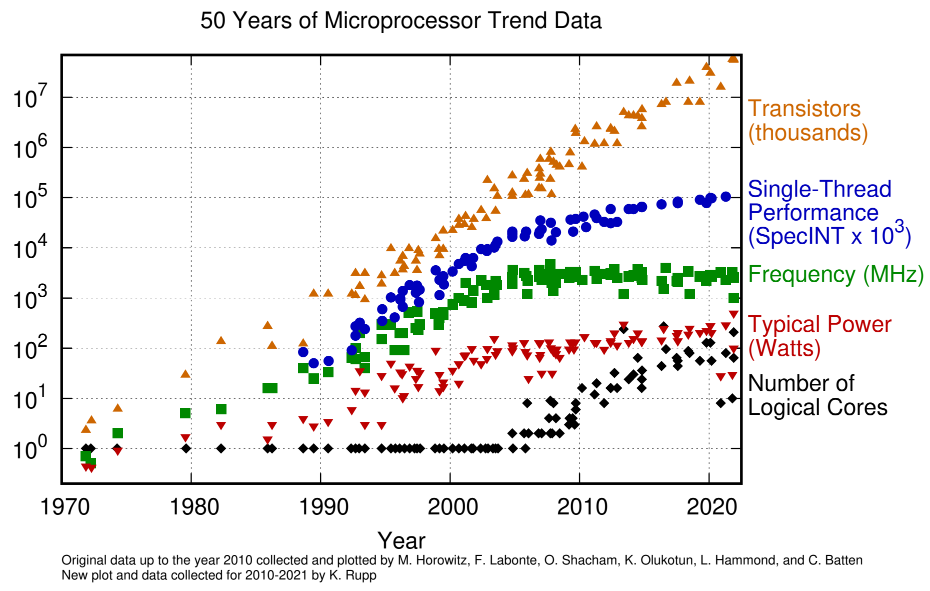


Figure 2.14: Evolution of number or Transistors, Logical Cores, Power consumption, Frequency and Single-Thread Performance since 1970 to 2021. Extracted from [22].

Parallelization can be achieved at different levels:

- **Instruction-Level Parallelism (ILP)**: this first level represent the most fine grain level of parallelization. It utilizes the capability of the processor to execute multiple instructions from a single thread. Some of the hardware techniques that allow this level are multi-stage pipelining, superscalar execution or Out-of-order execution.

- **Thread-Level Parallelism (TLP)**: at a higher level, this approach focus on the threads containing multiple instructions. This approach can be accomplished by simultaneous Multi-threading (SMT), where multiple threads utilize the same core and share its resources. Also, it can utilize multi-core architectures to offload and spread different threads in different cores. This complex task brings up new potential problems like tread interference (e.g. cache corruption or synchronization) the balancing of the different tasks and the synchronization among threads in bigger tasks and the possible overheads that can appear from them. Software platforms have been developed over the years to better support and give developers more control and tools when trying to archive this kind of parallelism. Some popular and open examples are OpenMP or Clink.

- **Data or Vector Parallelism**: this level utilizes instruction or more exactly data level granularity in order to perform one operation over multiple data elements simultaneously. This kind of parallelism requires comprehensive hardware support such as architectures capable of executing Single Instruction, Multiple Data (SIMD) instructions. Furthermore, Graphic Processing Units (GPUs) exploit this concept with massively parallel data interfaces and vector processing units capable of operating at great speed and efficiency with huge bandwidths. This level combines some of the problems of ILP and TLP but enables a new degree of freedom that makes it another powerful improvement for computing systems creating its own paradigm.

2.4.1 NUMA regions

Memory storage is one of the key components in modern computing systems due to its direct impact on computing power causing possible bottlenecks or enabling further improvements. Complex multi-core systems require memory modules of different sizes and access for optimal performance. However, the way these modules are connected to the rest of the system can vary. Shared memory architectures describe how the processors and the main memory units are placed, having two main strategies, uniform and non-uniform memory access. Uniform Memory Access (UMA) architecture defines a type of interconnection in which the memory access time is the same across all processors. Meanwhile, NUMAs (Non-Uniform Memory Accesss) shared memory architecture, the access time of the different Processing units is different as they possess their local memory that can be directly accessed but they can also access other memory modules that are local to other cores. These different architectures are showcased in Figure 2.15.

NUMA systems have proven in the recent years its advantages improving average case access time in hierarchical shared memory [23]. But this more complex architecture leads to potential problem when managing performance in NUMA sytems, the core affinity and the data placement.

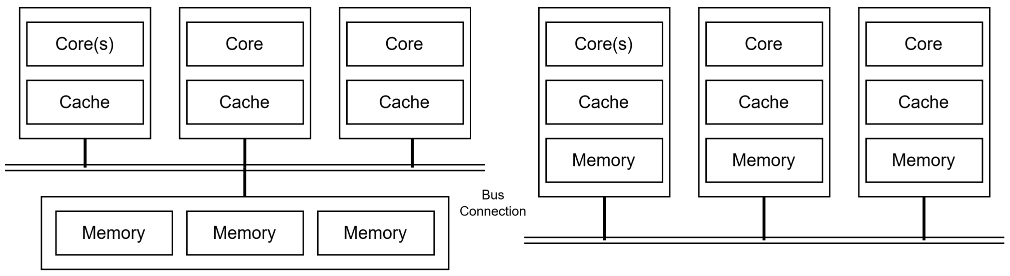


Figure 2.15: Simplified diagram of UMA (left) and NUMA (right) architectures in a multicore system

Due to the nature of operating systems and multi-threaded executions, a scheduler that is capable of assigning new tasks/threads to the different cores is required. This assignment can be ruled by other objectives, such as load balancing (spreading the threads among the system) or thread concentration (assigning threads to a few cores and putting the rest to low-power states). In NUMA systems the same hardware design imposes new properties such as the affinity among cores or their relative latency to neighbouring memory units.

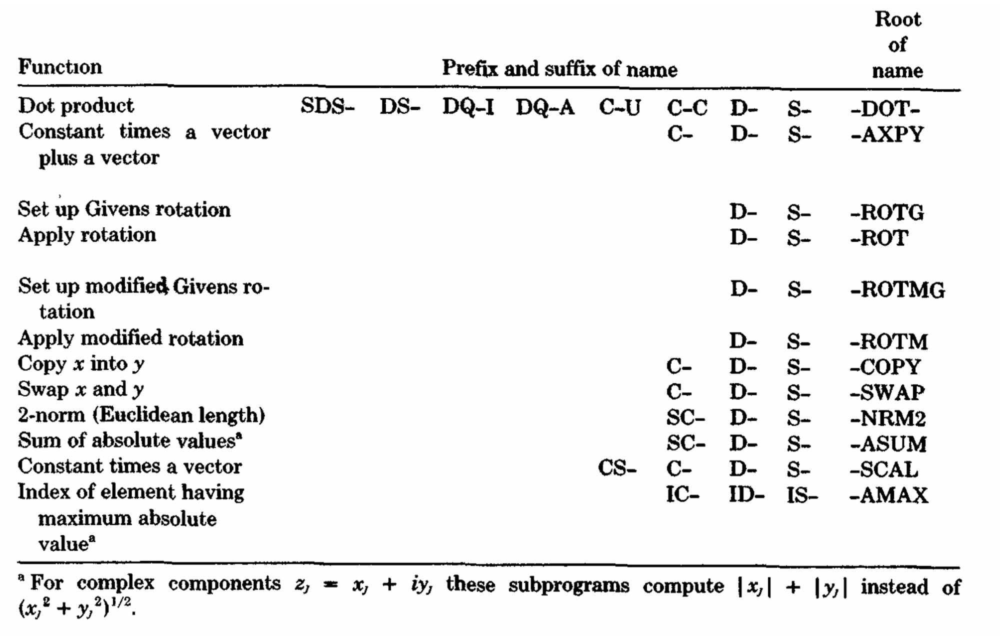
In the case of load balancing, one of the most common and default policies, thread migration can not take into account the local memory allocations and therefore generating a great potential overhead in data migration of access of the compared to the original location. This also reflects the importance of data placement and memory allocation. In scenarios where a multi-threaded application spawns more threads from the main one, the allocation of new memory can be crucial for the application's performance. There exist ways to define affinity and memory policies related to the core *libnuma* library for Linux and other APIs such as OpenMP.

2.5 Basic Linear Algebra Subprograms

Many computer applications in the realm of science, data science or graphics among others, rely heavily on linear algebra operations for many of its tasks. Therefore, there is a need for subroutines that are capable of doing this task in a reliable and efficient way so that more complex applications can be built on top of them with the complexity of supporting different software and hardware platforms.

2.5.1 Development and basic elements

Firstly born in the 1970s, the development of BLAS saw its first appearance as a unified work in "Basic Linear Algebra Subprograms for Fortran Usage" [24]. The improvement in computer science and the increasing complexity of programs at the time provoked industry software developers and researchers to develop low-level subroutines for these operations. This research work had the aim of unifying and standardising how these subroutines should be defined (Fig. 2.16), speeding up the software development process and generating portable structures and concepts that could be applied to other languages and platforms.



Function	Prefix and suffix of name								Root of name
Dot product	SDS-	DS-	DQ-I	DQ-A	C-U	C-C	D-	S-	-DOT-
Constant times a vector plus a vector						C-	D-	S-	-AXPY
Set up Givens rotation							D-	S-	-ROTG
Apply rotation							D-	S-	-ROT
Set up modified Givens rotation							D-	S-	-ROTMG
Apply modified rotation							D-	S-	-ROTM
Copy x into y						C-	D-	S-	-COPY
Swap x and y						C-	D-	S-	-SWAP
2-norm (Euclidean length)						SC-	D-	S-	-NRM2
Sum of absolute values[a]						SC-	D-	S-	-ASUM
Constant times a vector					CS-	C-	D-	S-	-SCAL
Index of element having maximum absolute value[a]						IC-	ID-	IS-	-AMAX

[a] For complex components $z_j = x_j + iy_j$ these subprograms compute $|x_j| + |y_j|$ instead of $(x_j^2 + y_j^2)^{1/2}$.

Figure 2.16: Summary of functions and name of BLAS Subprograms. Prefix letters I, S, D, C, Q denote the data type of the operation being integer,single-precision, double-precision, (single-precision) complex and extended precision, respectively. Extracted from [24].

Another key aspect of this first implementation is the programming language of choice, Fortran. This language, unlike C language, uses column-major storage which affects how the algorithms and their implementation. In subsequent years, the BLAS concept gained great popularity and expanded through the research world, being integrated in a library called LINPACK and standardised by the BLAS Technical Forum. The next step up in this direction was the extended capabilities with matrix-related operations. BLAS define 3 different levels of operations:

- **Level 1**: defined in the original paper it consists of vector-vector and vector-scalar operations:

$$y \leftarrow \alpha x + y$$

x, y being vectors and α a constant value.

- **Level 2**: the first expansion of the original work included more complex operations [25], based on matrix-vector operations, including general product, symmetric or Hermitian products, and triangular matrix-vector product. For the most relevant incorporation, the general matrix-vector product there exist 3 possible implementations depending on the value of the parameter "TRANS" given the function GEMV:

$$\text{if TRANS} = \text{'N'},\ y \leftarrow \alpha A x + \beta y$$
$$\text{if TRANS} = \text{'T'},\ y \leftarrow \alpha A^T x + \beta y$$
$$\text{if TRANS} = \text{'C'},\ y \leftarrow \alpha \overline{A}^T x + \beta y$$

x, y being vectors, α,β constant values, and A a matrix.

- **Level 3**: this final level targets the more complex matrix-matrix operations, providing a portable yet efficient implementation for computations such as matrix-matrix products, rank-k and rank-2k updates of symmetric matrix or triangular matrices operations [26]. Similarly to level 2, it provides specific implementations for the different possibilities of the main matrix-matrix product or more specifically multiply-add operation:

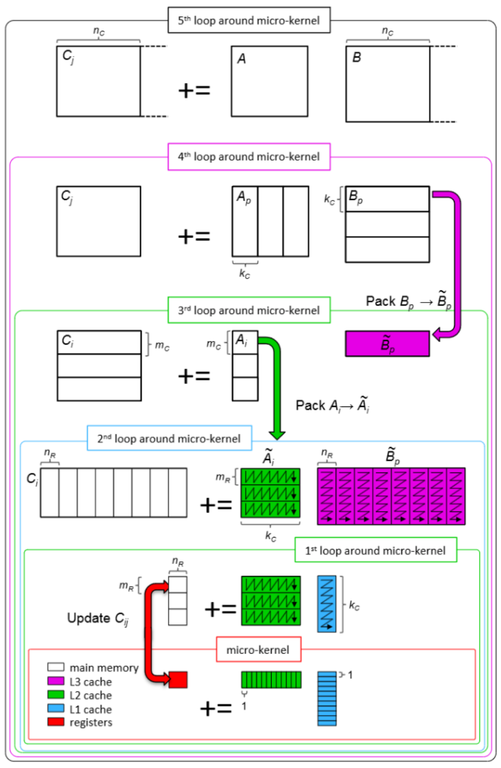
$$C \leftarrow \alpha AB + \beta C$$
$$C \leftarrow \alpha A^T B + \beta C$$
$$C \leftarrow \alpha AB^T + \beta C$$
$$C \leftarrow \alpha A^T B^T + \beta C$$

A,B,C being matrices and α, β constant values

With the introduction of level 2 and level 3 expansions in 1988 and 1990, relevant topics in high-performance computing were brought up like optimization of data movement in hierarchical memory systems and parallel processing of workloads. With these implementations included in the LINPACK library, many relevant companies developed versions for the computer architecture x86 with Intel MKL or IBM ESSL. LAPACK appeared as an open library that exploits the developments in BLAS in libraries such as LINPACK but its main target was running efficiently on shared-memory and hierarchical systems and even vector/parallel processors, unlike LINPACK or EISPACK (another popular BLAS library). This new library was also developed in Fortran and offered APIs in C standard language to be used in different scenarios.

Automatically Tuned Linear Algebra Software (ATLAS) [27] at the end of the 90s tackled the problem of generalizing the BLAS implementations to different hardware while being aware of their deep memory hierarchies, generating automatically optimized code for level 3 BLAS implementations (mostly for DGEMM). This automation seeks to reduce the time of modifying the already existing libraries for Linear Algebra to better suit a great variety of architectures and hardware types, outperforming the vendor-provided code in many cases, at that time. This strategy not only relied on hardware knowledge beforehand but also empirically tested different elements of the underlying silicon. Different code versions (e.g. loop reordering, different tiling sizes) at a high level are tested to then obtain the optimal parameters to specifically tune an on-chip multiply that can be used to build a complete matrix-matrix multiply.

The next biggest achievement and what became one the foundations of many modern BLAS and HPC libraries, such as OpenBLAS or Intel MKL, was the appearance of the GotoBLAS algorithms in the revolutionary paper "Anatomy of high-performance matrix multiplication" [28]. This paper took all the previous knowledge to the moment and created a set of algorithms that used layering and tiling with specific hardware-aware sizes of different level memories (that in this case had to be predefined) together high low-level optimized kernels for the most inner loops of execution. It highlighted the importance of higher-level cache utilization and data locality and continuity as well as the importance of register-level sizes for the dimension of the specialised inner kernels. The knowledge of all of these properties and the different layers of optimization can be seen in Fig. 2.17. This optimization also included multi-thread execution and awareness in order to potentiate the performance in more complex machines.


5th loop around micro-kernel
n_C
C_j
+=
A
B
4th loop around micro-kernel
A_p
B_p
k_C
Pack $B_p \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_C
A_i
$\tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
n_R
m_R
$\tilde{A}_i$
1st loop around micro-kernel
Update C_{ij}
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

```
Loop 5  for $j_c = 0 : n-1$ steps of $n_c$
          $\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4    for $p_c = 0 : k-1$ steps of $k_c$
            $\mathcal{P}_c = p_c : p_c + k_c - 1$
            $B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$          // Pack into $B_c$
Loop 3      for $i_c = 0 : m-1$ steps of $m_c$
              $\mathcal{I}_c = i_c : i_c + m_c - 1$
              $A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$        // Pack into $A_c$
              // Macro-kernel
Loop 2        for $j_r = 0 : n_c - 1$ steps of $n_r$
                $\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1          for $i_r = 0 : m_c - 1$ steps of $m_r$
                  $\mathcal{I}_r = i_r : i_r + m_r - 1$
                  // Micro-kernel
Loop 0            for $k_r = 0 : k_c - 1$
                        $C_c(\mathcal{I}_r, \mathcal{J}_r)$
                             $+= A_c(\mathcal{I}_r, k_r) \; B_c(k_r, \mathcal{J}_r)$
                  endfor
                endfor
              endfor
            endfor
          endfor
        endfor
```

Figure 2.17: "Left: The GotoBLAS algorithm for matrix-matrix multiplication as refactored in BLIS. Right: the same algorithm, but expressed as loops". Extracted from [29].

Chapter 3

Related Work

3.1 HPC Platforms

The field of HPC has been in the focus of many companies and research due to the increasing demand of cloud computing and heavy computational loads such as data analytics, machine learning or scientific simulations. With this in mind, over time different options have emerged trying to leverage the benefits of RISC-V to push the limits of computation.

3.1.1 Manticore

In 2020, one of the most ambitious projects related to RISC-V cores and high performance with efficiency saw the light, the Manticore chiplet [30]. This work develops a 4096-core RISC-V chiplet architecture with a focus on ultra-efficient Floating-point Computing. This project focused on two pivotal points:

Chiplet architecture

This novel design used the chiplet structure to improve the cost and yield of the manufacturing cost. For a single full chip, it uses 4 different chiplets, each of them composed of 32 clusters with eight 32-bit RISC-V processor cores based of the Snitch [31] design. Furthermore, every chiplet includes a 64-bit RISC-V Ariane/CVA6 [32] cores for management, 27MB of shared L2 memory, a HBM (High Bandwidth Memory) controller, a 16x PCIe endpoint for host communication and a private 8GB HBM. There exist also in-package chip-to-chip links, one to each sibling for inter-die synchronization and chiplet-to-chiplet NUMA (Non-Uniform Memory Access).

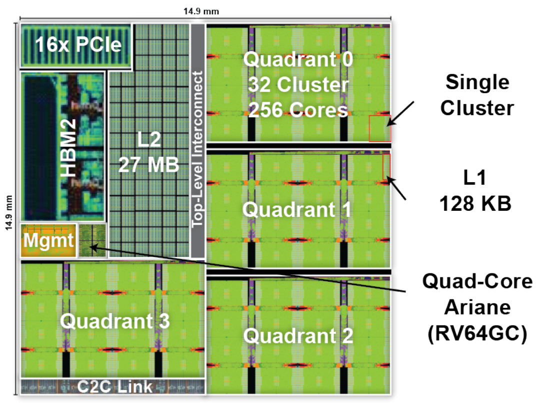


Figure 3.1: "Conceptual floorplan of the package. Arrangement of the chiplets and HBM on the interposer. Each chiplet has its own, private, 8 GB HBM. Chiplets interconnect via die-to-die serial links". Extracted from [30]

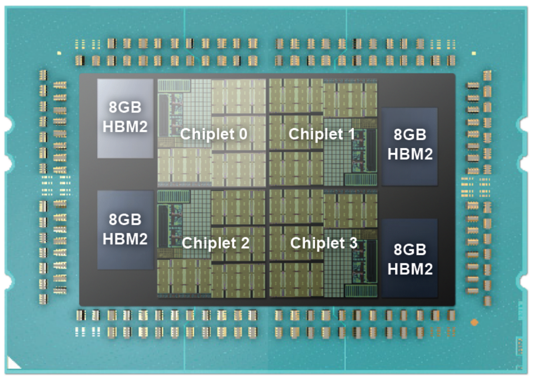


Figure 3.2: "Conceptual floorplan of an individual chiplet. Arrangement of individual cluster quadrants, interconnects, L2 memory, HBM2 controller, PCIe controller, and quad-core Ariane RV64GC system". Extracted from [30]

Custom ISA Extensions

The cores used in the design include the basic RISC-V base and general extensions with single and double-precision floating-point arithmetic and two fully custom ISA extensions in order to improve the utilization and performance of the floating-point unit. The Stream Semantic Registers (SSRs) extension target is to reduce the large number of loads/stores instructions by explicitly encoding memory accesses as register/read writes, which requires to give a subset of the processor core's registers stream semantics [30] [33]. The second of these extensions is the Floating-Point Repetition (frep) which implements a FPU (Floating-Point Unit) exclusive hardware loop where "hot" micro-loops can be executed in a sequence buffer that works independently of the main Snitch cores. This accelerates this kind of very common sub workloads like in vector-matrix dot product with floating-point numbers [30]. Fig. 3.3 and Fig. 3.4 showcase an examples of the optimizations and implementation of both SSRs and frep and how they can improve the execution of the baseline RISC-V instructions.

With all of this elements, the Manticore project accomplished to compete with NVIDIA V100 GPU in performance efficiency for DNN training tasks such as Convolution reaching more than 50 GFLOP/s per Watt [30], even outperforming many high-end high-performance CPUs such as the Intel i9-9900K or the Neoverse N1.

```
loop:                          scfg 0, %[a], ldA
fld f0, %[a]                   scfg 1, %[b], ldB
fld f1, %[b]      ────────►    loop:
fmadd f2, f0, f1               fmadd f2, ssr0, ssr1
```

Figure 3.3: "Left: baseline simplified RISC-V implementation, with address calculation and pointer increment omitted for brevity.Right: SSRs implementation with memory loads encoded as reads from stream registers; additional stream configuration instructions required ahead of the loop.". Extracted from [30]

```
mv    r0, zero
loop:                                 frep  r1, 1
addi  r0, 1             ────────►     loop:
fmadd f2, ssr0, ssr1                  fmadd f2, ssr0, ssr1
bne   f0, r1, loop
```

Figure 3.4: "Left: implementation with loop bookkeeping using baseline RISC-V instructions. Right: implementation with an frep hardware loop, with all bookkeeping to occur implicitly in the hardware". Extracted from [30]

3.1.2 Xuantie 910

Presented in paper "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension" [34] this newer CPU architecture builds upon the academic work of other RISC-V CPUs such as BOOM [35] and other simpler cores like RI5CY [36] and other industrial work from vendors such as SiFive or Microsemi in order to create a high-performance multicore CPU based on 64-bit RISC-V. The company Alibaba T-Head is the designer of this hardware that uses a multi-cluster architecture, where each single cluster can have 1,2 or 4 cores. Each of these cores is based on the XT-910 architecture running at up to 2.5GHz and supports a 32/64 KB L1 instruction and data cache. Then each cluster shares a 8/16 way associated L2 cache with up to 8MB of memory capacity.

XT-910 core architecture

The XT-910 architecture developed for this CPU is compliant with the RISC-V specification RV64GC, the Vector extension on its 0.7.1 version and other non-standard extensions. The main elements of the core are:

- **12-stage Out-of-order pipeline**: the "front end" is composed by 7 stages, including instructions fetch, decode and Issue. Then we find 4 different branches for execution: branch jump pipe, two scalar floating point units and two vector execution unit, one dual-issue out-of-order load & store unit. The out-of-order issue engine can pass up to eight instructions.
- **Vector Unit and Extension**: The vector execution pipeline present in this architecture is based on the draft stable release version 0.7.1 of the RISC-V

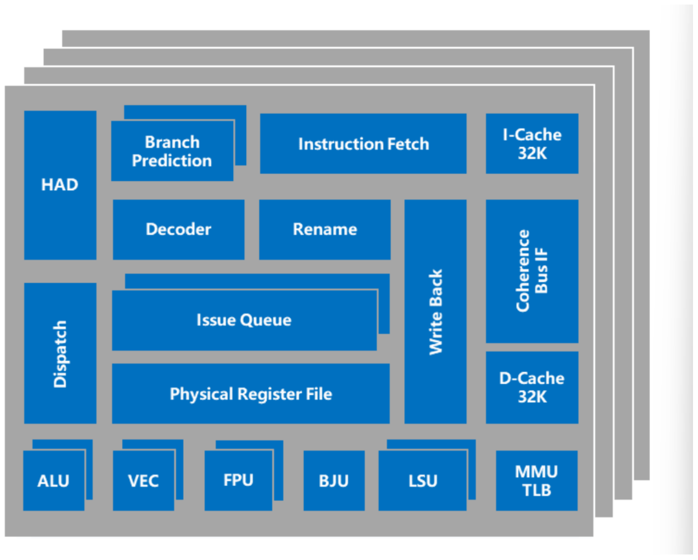


Figure 3.5: Diagram of XT-910 multi-core cluster (4-core configuration). Extracted from [34].

Vector specification. It is composed of multiple identical 64-bit scalar pipelines that support 8-bit to 64-bit vector integer operations, half-precision/single-precision/double-precision floating. This pipelines are divided in vector slices, which have a complete 64-bit data path, 64-bit vector physical register file and two out-of-order vector integer and floating-point units. This results on a total of 256-bit operation results in one clock cycle and a 128-bit vector/load store operation.

- **Memory management**: The architecture includes a SV39 MMU being compliant with RISC-V Linux specification. It includes also support for huge-page mapping and multi-size (4K,2M and 1G) entry at all levels of TLB (Translation Lookaside Buffer), allowing for a iterative process through the different sizes to reduce page misses.

- **Custom extensions**:in order to improve performance, T-Head designed different non-standard custom extensions that add more than 50 non-standard instructions. These accelerate domain-specific tasks and include arithmetic operations, load & store, TLB, bit manipulation, cache operations, expanded MMU management.

Along with these specifications and novel hardware, Alibaba T-Head developed a software ecosystem that includes a full compilation toolchain based on the official GNU tools and an IDE that enables RISC-V graphical trace, profiling and instruction accurate simulation with JTAG online debug. This allows for a better

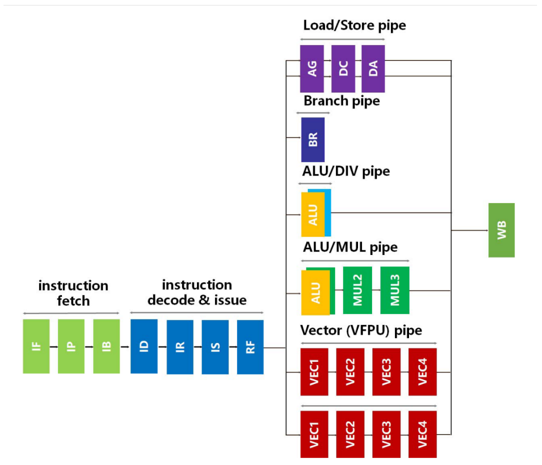


Figure 3.6: 12-stage pipeline in XT-910 core. Extracted from [34]

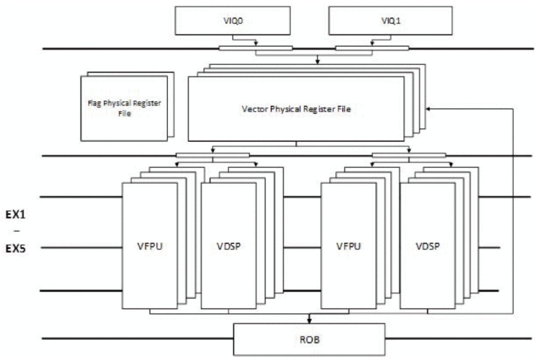


Figure 3.7: Pipelined Vector operation architecture. Extracted from [34]

matching between hardware and software as these toolchains have been developed to support all the hardware characteristics and optimise software execution to them. All in all, the Xuantie 910 established itself as a commercial competitor to similar 64-bit cores with other more classical ISAs such as the popular ARM cortex A series (compared in [34] with the ARM Cortex-A73) and a flagship design inside of the RISC-V high-performance cores world.

3.1.3 Milk-V Pioneer

Many companies have emerged in the last years with the developments on RISC-V and newer hardware platforms. Milk-V is one of this companies that is committed to provide new RISC-V products, embracing its open-source and community-based philosophy. Among its broad catalogue one of the most powerful platforms is the Milk-V Pioneer, which targets to make native RISC-V development possible through a high-performance CPU equipped with state-of-the-art complementary hardware and enabling possible expansions as can be seen in Fig. 3.8. At the centre of this platform we can find the Sophon SG2042, a cutting-edge multi-core CPU based on RISC-V.

Sophon SG2042

The Sophon 2042 represents a big step forward in the commercially available mass-produced CPUs based on RISC-V Cores. It utilises 16 multi-core Xuantie C920, clusters composed by 4 cores, organised and connected into a mesh network

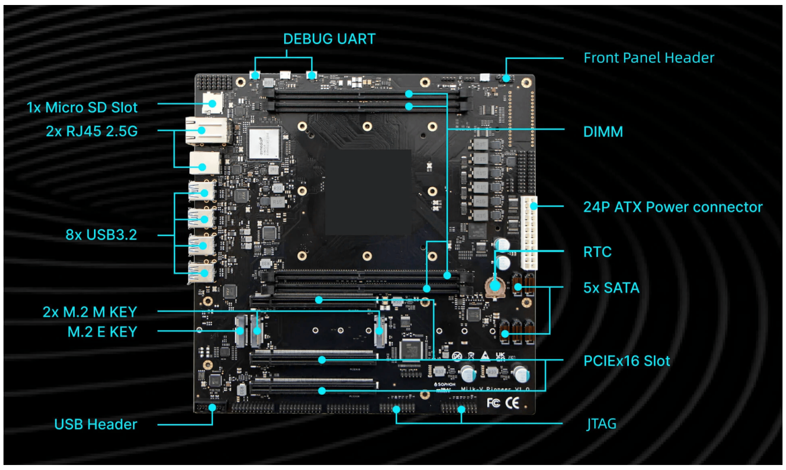


Figure 3.8: Milk-V Pioneer motherboard and main hardware characteristics. Extracted from [37].

architecture or NoC (Network-on-Chip) architecture, displayed in the diagram of Fig. 3.9. Each system Level Cache is 4MiB in size totally and 16 of them are connected in the network, corresponding to 64MiB system level L3 cache. Additionally it includes four DRAM controllers) (supporting DDR4 UDIMM/SODIMM/RDIMM up to 3200MT/s with ECC byte) located on the left and right side respectively, accessible by all masters in the network. Finally thre are two more components, the PCI devices and the System CoProcessor, whose task is to coordinate and initialize basic platform elements such as the DRAM controller, PCIe Controller or mesh setup. All of these components and the extra CCIX ports for 2 sockets mode are represented in Figure 3.9. The Xuantie 920 utilizes pretty similar configurarions and characteristics of its open-source companion, the C910. The main characteristics of this core are:

- 64-bit core implementing the standard extensions IMAFDC.
- Incorporates the RISC-V vector extension in its stable draft version 0.7.1.
- Runs at operation frecuency of 2.0GHz
- Uses a 64KiB L1 I-Cache and 64KiB L1 D-Cache per core configuration and 1MiB unified L2 cache per cluster

3.2 Inference frameworks

The deployment and execution of LLMs require a piece of software capable of interpreting the model format and adding a computational structure, these are

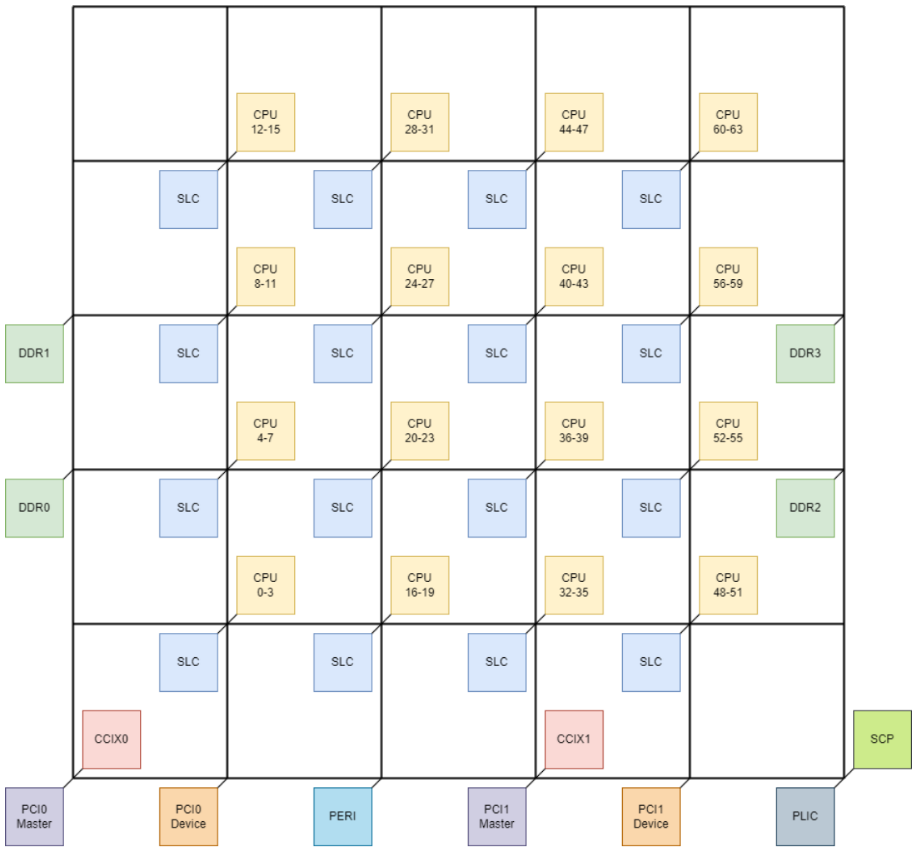


Figure 3.9: SG2042 Mesh architecture and main components. Extracted from [38].

the inference frameworks. Machine Learning models are built and require tensor libraries as an efficient and flexible way to provide fundamental data structures and operations necessary in ML. These building blocks encapsulate tensor (multi-dimensional arrays) operations and optimised versions for linear algebra, including sometimes hardware-specific implementations. Built atop tensor libraries, we can find training and inference frameworks that provide high-level APIs and tools to interact with the models.

Training frameworks include functionalities for designing, training and evaluating the performance and evolution of the models. For these tasks, it is common to find an array of tools that are not typically available in specialized inference frameworks. These may include optimization algorithms, automatic differentiation, as well as utilities for dataset management, pre-processing, and augmentation, among others. On the other hand, the specific inference frameworks focus on performance and resource efficiency. Here, for a definitive model different optimizations can be done at the graph level such as operation fusion, in the implementation level, providing a code that performs the node's task with a hardware-specific version or even improving memory bandwidth thanks to quantization techniques. These tools are not exclusive and there exist all-in-one or more complete solutions.

3.2.1 llama.cpp and ggml

Two notable open-source repositories that gained notable attention recently in this field are llama.cpp and GGML. Generative Generalized Machine Learning (GGML) is the backbone tensor library that llama.cpp utilizes to provide fast inference with full flexibility and simplicity provided in C/C++.It comes by default with many implementations for more established CPUs and ISAs such as x86 and ARM, as well as support to their vector extensions by specific implementations. Furthermore, the only optimized targets are not desktop CPUs, during the building process it can include other backends such as CUDA, BLAS libraries or Vulkan to target GPUs or make hybrid computation offloading certain layers to other hardware.

GGML main elements

As described previously, GGML library defines the different data structures and operations that comprise ML models and their execution. Inside this library we can observe the definition of the most elemental data element inside GGML as shown in the code fragment 3.1 .

Listing 3.1: simplified ggml_tensor definition extracted from ggml.h [39]

```
struct ggml_tensor {
        enum ggml_type type;
        struct ggml_backend_buffer * buffer;
...
        int64_t ne[GGML_MAX_DIMS]; // number of elements
        size_t  nb[GGML_MAX_DIMS]; // stride in bytes:
        // nb[0] = ggml_type_size(type)
        // nb[1] = nb[0]   * (ne[0] / ggml_blck_size(type)) + padding
        // nb[i] = nb[i-1] * ne[i-1]
        // compute data
        enum ggml_op op;
        // op params - allocated as int32_t for alignment
        int32_t op_params[GGML_MAX_OP_PARAMS / sizeof(int32_t)];
        int32_t flags;
...
        // source tensor and offset for views
        struct ggml_tensor * view_src;
        size_t               view_offs;
        void * data;
        char name[GGML_MAX_NAME];
        void * extra; // extra things e.g. for ggml-cuda.cu
};
```

Inside this structure, there are many parameters with different functions but the essential ones are:

- **ggml_type**: this enumeration defines the type of data of the individual elements that will be contained in the tensor. Some classic datatypes like single-precision floating-point are represented as "GGML_TYPE_F32" but more complex types are also included, as different quantized data types for different lengths (e.g. 8-bit quantization GGML_TYPE_Q8_0 vs 4-bit quantization GGML_TYPE_Q4_0) or different quantization methods (e.g. 8-bit k-quants GGML_TYPE_Q8_K vs 8-bit "type-1" quantization GGML_TYPE_Q8_1).

- **ne**: it refers to the number of elements in different dimensions. This element allows to calculation of indices and moves through the tensors. GGML, as C language, uses row-major ordering, therefore the value of "ne[0]" will correspond to the number of elements in each row, "ne[1]" will store the column size and this structure goes on up to "GGML_MAX_DIMS", which in ggml is equal to 4.

- **nb**: in this variable is stored the stride in bytes between numbers inside of each of the same dimensions. This is a complex variable as depending on the number format the stride can vary largely. It can be computed in the following as specified in lines 7-9 of code fragment 3.1

$$\begin{aligned} nb[0] &= ggml\_type\_size(type) \\ nb[1] &= nb[0] \times (ne[0]/ggml\_blck\_size(type)) + padding \\ nb[i] &= nb[i-1] * ne[i-1] \end{aligned} \tag{3.1}$$

  For classic non-quantized datatypes with no special padding, and a classic tensor, all the strides to navigate through the tensor can be obtained with the first and last equations. But quantized models use the concept of blocks, for which the numerical quantized values have to be dequantized before being used in order to obtain the true value. These extra required parameters are usually also stored as part of the tensor and therefore, there is the necessity of differentiate between intra-block (where values are contiguous with their own stride depending on the datatype) and inter-blocks (considering the extra bytes of non-values).

- **op**: the operation parameter defines which will be the use of the tensor. When a new tensor operation is required, the result tensor is created with the operation encoded and the sources of data included through the src pointers in the structure.

Despite the apparent complexity of the core parameters in the tensors, they provide significant benefits, mainly allowing the storage of non-continuous tensors and simplification of tasks like transposition and permutation, which only require pointers resignation, but not actual data movement.

The library offers the freedom to the user to generate its own functions using available tensor operations. The functions are represented internally as computation graphs and each of the operations inside of the graph corresponds to a node. Once the different steps in the computation are defined through the ggml tensors and the inputs are defined it can be executed. For the sake of clarity in the code 3.2 is showcased the example of how a function corresponding to the mathematical expression $f(x) = a * x^2 + b$ is created and executed using the ggml library:
3.1

Listing 3.2: modified example of definition and execution of function $f(x) = a * x^2 + b$, extracted from ggml.h [39]

```
{
        struct ggml_init_params params = {
            .mem_size   = 16*1024*1024,
            .mem_buffer = NULL,
        };

        // memory allocation happens here
        struct ggml_context * ctx = ggml_init(params);

        struct ggml_tensor * x = ggml_new_tensor_1d(ctx, GGML_TYPE_F32
    , 1);

        ggml_set_param(ctx, x); // x is an input variable
         //Definition of the tensors that will participate in the
    fuctions and its relationship with the input values
        struct ggml_tensor * a  = ggml_new_tensor_1d(ctx,
    GGML_TYPE_F32, 1);
        struct ggml_tensor * b  = ggml_new_tensor_1d(ctx,
    GGML_TYPE_F32, 1);
        struct ggml_tensor * x2 = ggml_mul(ctx, x, x);
        struct ggml_tensor * f  = ggml_add(ctx, ggml_mul(ctx, a, x2),
    b);

        struct ggml_cgraph * gf = ggml_new_graph(ctx);
        ggml_build_forward_expand(gf, f);

        // set the input variable and parameter values
        ggml_set_f32(x, 2.0f);
        ggml_set_f32(a, 3.0f);
        ggml_set_f32(b, 4.0f);

        //Execution of the contructed computation
        ggml_graph_compute_with_ctx(ctx, &gf, n_threads);

        printf("f = %f\n", ggml_get_f32_1d(f, 0));
};
```

3.3 Compilers for RISC-V

A key component in computer science are compilers, doing the crucial task of translating high-level programming languages, that we humans can use more easily into machine code that can be executed by computers and processors.

Two of the most popular compilers worldwide currently are GCC C and Clang when we talk about C, one of the most important and used languages in the history of computer science. GCC C belongs to the GNU Compiler Collection, while Clang is part of the LLVM project. Both of them have support for multiple computer architectures and even custom extensions for better usage in the underlying hardware. Both of them are part of community-based development and are completely open-source, but they have many differences.

3.3.1 GCC

In 1984 the GNU project was given birth to become a UNIX-like open-source software system. Inside of this group of tool there were some of the ones that today constitute part of the backbone of the GNU/Linux ecosystem such as GlibC, GDB, Make or GCC. Originally born as a C compiler evolved as a full collection of compilers to support other languages like Fortran or Objective-C. The development of GCC has been slow, suffering from the split in effort during of its early years between the GCC and GNU system, in 2015 with GCC-5.0, the policy and rhythm of development changed radically, being now a priority to develop a competitive compiler that would deliver a major version release each year, in order to keep up with the newer alternatives.

The architecture of GCC follows the structure of a monolithic compiler with several elements and representations working tightly together. GCC utilises several intermediate representations with different levels of optimisation. Firstly, the code is turned into a an standard generic abstract syntax tree (AST). This representation, due to language differences, can change in format but the AST is turned into a unified form called generic. After the end of the front end, the generic representation is converted into the GIMPLE form, which focuses on high-level elements and it is useful for optimization of the source code. Then, it is turned into a static single assignments (SSA) representation. In this form, several optimizations and versions of the variables depend on their usage, operation and possible branches. In this shape, GCC performs more than 20 different optimisations. After all the optimizations the SSA shape is transformed back to GIMBLE representation.

At the lowest level of optimizations and representations, this GIMBLE representation is turned into an RTL tree representation. In this form, which is hardware-based and utilizes the abstract target architecture, further improvements are produced. And finally, the GCC back end generates the assembly code necessary to generate the machine code to run on a specific architecture.

All of these steps can be visualized at the diagram of Fig. 3.10.

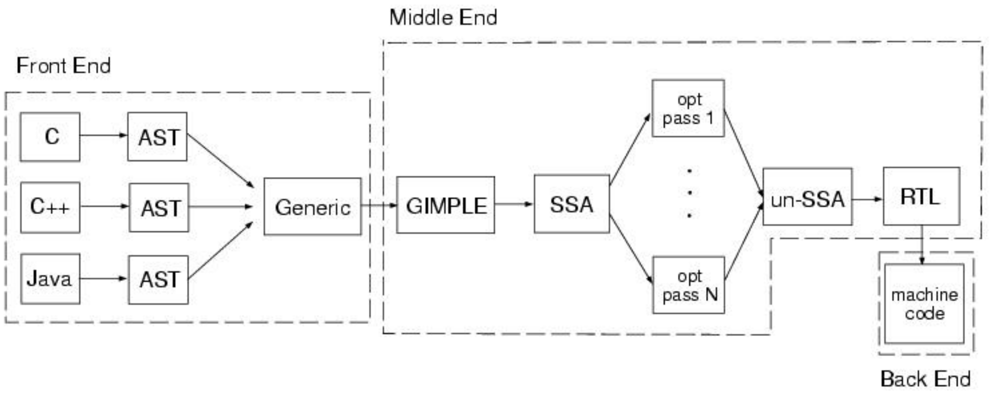


Figure 3.10: GCC execution diagram with front, middle and back ends and intermediate representations. Extracted from [40].

3.3.2 CLANG/LLVM

Born in 2003, the Low-Level Virtual Machine (LLVM) was developed as a modular and flexible framework for developing compilers. Upon this software infrastructure, Clang was born as an alternative to GCC C compiler, improving in many areas that GCC was lacking behind. LLVM allows developers to separate the two main sections, the front end and the back end.

This allows higher-level optimization and while being easy to support different architectures. Another big improvement compared to GCC, is the improved diagnostics included in the same compiler. This allows for faster development and debugging for the developers. The next differentiating factor is extensibility, LLVM was written to ease the development of new extensions and support for newer modules that can integrate easily.

Regarding the execution and structure itself, it is also quite different compared to GCC. The main difference and distinct element of LLVM is the LLVM Intermediate

Representation (IR). This is a mid-level representation that can also come from an AST representation. This mid-representation is also based on SSA representation and uses a specific language and structure that focuses on supporting lightweight runtime optimizations and restructure transformations among others. This intermediate representation is the one over which the optimizations are performed as shown in Fig. 3.11.

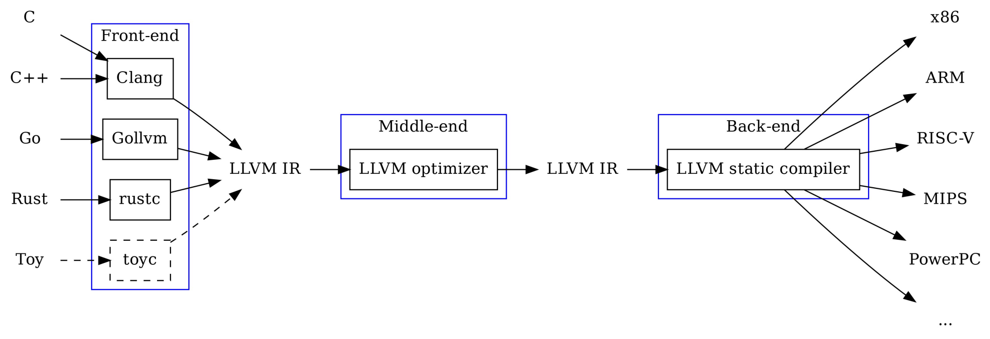


Figure 3.11: LLVM execution diagram with front, middle and back ends and intermediate representations. Extracted from [41].

All in all, LLVM is a modern compiler that has proven in different computer architectures to be able to extract more performance, showing a better performance in many comparisons to GCC.

3.4 LLM models

After the appearance of the Transformers architecture in 2017, bigger and bigger models, trained in billions of data have been developed and have pushed the capabilities of Deep neural networks in the field of natural language processing (NLP). At the front of this revolution, some companies such as OpenAI with its GPT family and Google models like BERT paved the way for newer and more powerful models, with larger massive datasets and increased training times. This race towards bigger models and improved training is still nowadays a reality that involves many research facilities and companies.

3.4.1 Meta LLAMA

In February of 2023 Meta AI published a revolutionising paper called "LLaMA: Open and Efficient Foundation Language Models" [42]. This paper brought to the table a new perspective, providing a new collection of foundation language models ranging from 7 billion to 65 billion parameters. At the time, OpenAI's

GPT-3 and Google's PaLM already showcased impressive and state-of-the-art results. LLaMa models not only proof a competitive performance in comparison but also a significant reduction in size and resources. Its approach relied on more specialized training completely over open source datasets like CommonCrawl or Wikipedia. Its main architectural characteristics and inspirations are:

- **Pre-normalization**: inpired by GPT3, it searched to improve training stability.
- **SwiGLU activation function**: Unlike the commonly used ReLu non-linear function, for these models it was replaced by the SwiGLU, as with the PaLM model but with a slightly different dimension.
- **Rotary Embeddings**: in this novel architecture it was used rotary positional embedding (RoPE) at each layer of the network, instead of the traditional absolute positional embeddings.

Meta AI released in an open-source manner the weights of their four models with 7B, 13B, 33B, and 65B parameters. This led to a huge world of opportunities for research and the open public as customizations and fine-tuning was now available for everyone on these models. Due to the worldwide success of LLaMa, Meta introduces in July of 2023 LLaMa 2 [43], in 3 different sizes (7B, 13B, and 70B parameters). The new improved generation followed the same philosophy of its predecessors and was open-source with commercial and research licences for complete usability and adaptability to other applications or explorations. New models included two main architectural novelties compared to the previous generation: the increased context length and the grouped-query attention (GDA).

One of the main targets, due to the new trends, was the option of having fine-tuned model versions that could be specialised in dialogue use cases. This led to the birth of the Llama-2-Chat collection of models. Based on the Llama 2 models, these models went through a specific fine-tuning involving human feedback and reinforced learning. The detailed process is explained in Fig. 3.12.

3.5 BLAS Libraries

Modern heavy computational workloads, especially those related to HPC, require extracting most of the performance with maximum efficiency from the hardware it runs on. From the exploitation of vector instructions to optimize data access in complex hierarchical memory systems. One of the most important operations that is the core of many applications such as AI or digital processing is matrix multiplication. This central operation and other close ones like vector-related operations have been the centre of many developments.

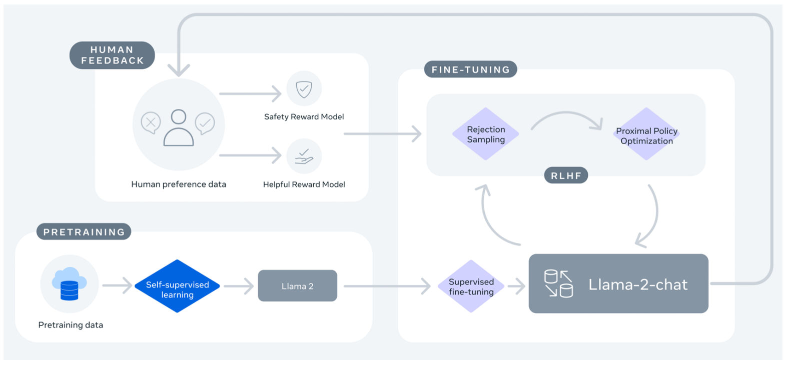


Figure 3.12: " Training of Llama 2-Chat: This process begins with the pretraining of Llama 2 using publicly available online sources. Following this, we create an initial version of Llama 2-Chat through the application of supervised fine-tuning. Subsequently, the model is iteratively refined using Reinforcement Learning with Human Feedback (RLHF) methodologies, specifically through rejection sampling and Proximal Policy Optimization (PPO). Throughout the RLHF stage, the accumulation of iterative reward modelling data in parallel with model enhancements is crucial to ensure the reward models remain within distribution". Extracted from [43].

3.5.1 OpenBLAS

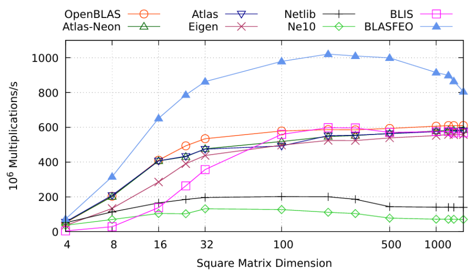
OpenBLAS represents the most important and spread open-source implementation of BLAS and LAPACK libraries. It provides APIs for level 1,2 and 3 routines in commonly single-precision and double-precision floating-point routines. Furthermore, it supports most of the main computer architectures with many CPU-specific optimizations.

In 2011, a team from the State Key Lab of Computing Science, Chinese Academy of Sciences in Beijing proposed an improved algorithm based on the GotoBLAS algorithm and LAPACK routines to boost the performance and give support to the Loongson 3A CPU, based on the MIPS64 microarchitecture [44]. To scale GEMM performances other memory-related optimizations are used. For instance, Cache and Register blocking, loop unrolling and reordering instructions and 128-bit memory accessing instructions, available in the Loongson 3A, and software prefetching. These memory optimizations can be performed thanks to the data stored in the core/architecture definitions, including the different sizes of memories number of DTB entries and line size as shown in the code snippet 3.3.

Listing 3.3: Information about LoongSon 3R3, 3A, 3B cache size and memory data, in get_target.c

```
#if defined FORCE_LOONGSON3R3 || defined FORCE_LOONGSON3A || defined
    FORCE_LOONGSON3B
#define FORCE
#define ARCHITECTURE     "MIPS"
#define SUBARCHITECTURE "LOONGSON3R3"
#define SUBDIRNAME       "mips64"
#define ARCHCONFIG    "-DLOONGSON3R3 " \
        "-DL1_DATA_SIZE=65536 -DL1_DATA_LINESIZE=32 " \
        "-DL2_SIZE=512488 -DL2_LINESIZE=32 " \
        "-DDTB_DEFAULT_ENTRIES=64 -DDTB_SIZE=4096 -DL2_ASSOCIATIVE=4 "
#define LIBNAME    "loongson3r3"
#define CORENAME   "LOONGSON3R3"
#else
#endif
```

This project was open-sourced and exposed to the community to as OpenBLAS, forked from the original GotoBLAS2-1.13 BSD version[45]. With time, community and research centres have further expanded the specific hardware support, with handmade customized assembler kernels. As up to date, it supports, Intel and AMD x86/x86-64, MIPS32/64, ARMv6/v7, multiple versions of the ARM64 microarchitecture, PPC/PPC64, IBM zEnterprise System and some RISC-V cores. It even includes the option of auto-recognizing the system architecture to choose the correct implementation. Furthermore, its native implementation of multi-threading, makes it suitable not only for single-core applications but for larger systems where concurrency and awareness about the problems of scalability are being tackled. This made it a real competitor even for less mature architectures such as ARM and RISC-V, as shown in real cases with different evaluations as the one example displayed in Fig. 3.13.



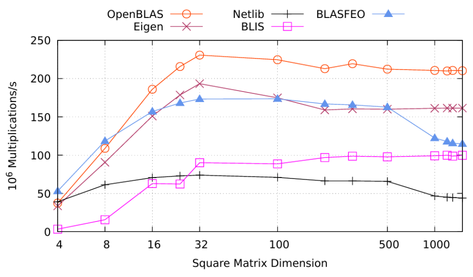


Figure 3.13: "SGEMM performance in multiplications per second, using SiFive RV64GCSU core and ARM Cortex-A9. Extracted from [46].

All of these options, open-source character and competitive performance make OpenBLAS a real viable option for many CPUs and architectures, with community support and constant maintenance.

Chapter 4

Methods

In the current chapter, we'll delve into the main objectives of this work, getting a better understanding of the tasks developed and the reasons behind them. This chapter will be divided into sections based on the different technical aspects and developments to improve the inference process in a RISC-V high-performance platform. The targeted HPC platform has been the MILK-V Pioneer computer, provided by the research team of Integrated Systems of ETH, equipped with the Sophon SG2042 CPU and its 64 RISC-V C920 Alibaba T-Head cores. This product marks one of the first major releases in the RISC-V HPC field. It provides cutting-edge high performance in a classical mainframe computer format. This computer runs a full-fledged Linux Desktop distribution, in this case, Fedora, with all the default available tools and software.

4.1 Exploiting hardware architecture

In the following sections we explain the two main areas that affect the efficiency and performance when working with the RISC-V cutting edge hardware used. The first subsection focuses on software ecosystem that is can be used natively in the machine, mostly compilers. Secondly we explore the hardware architecture through different the software tools avaliable in Linux and give crucial information for the usage of this complex machine.

4.1.1 Available compilers for the platform and support

One of the most important software elements to optimize the deployment and execution of a program is the compilation toolchain. Many companies search for the optimal framework and its configuration, to squeeze out every single performance capability out of their software. Due to the nature of RISC-V and its mature state in other embedded systems, the most common approach is cross-compilation instead of native compilation.

Most of the RISC-V embedded systems and even low-power CPUs systems like SBC systems, lack the memory requirements or the computational power to be able to compile code for their own platform, or in some cases, it is not the preferred choice due to impracticality. Cross-compilers allow the use of a different platform, commonly with a completely different architecture with respect to the target hardware, to execute all the compilation processes. Therefore, cross-compilation can be used to generate machine code executable by the target platform through the use of another machine. This process allows faster and better compilation and the cost of needing a bigger setup for the compiler due to the dependencies and extra libraries required. Our system, on the other hand, meets all the requirements

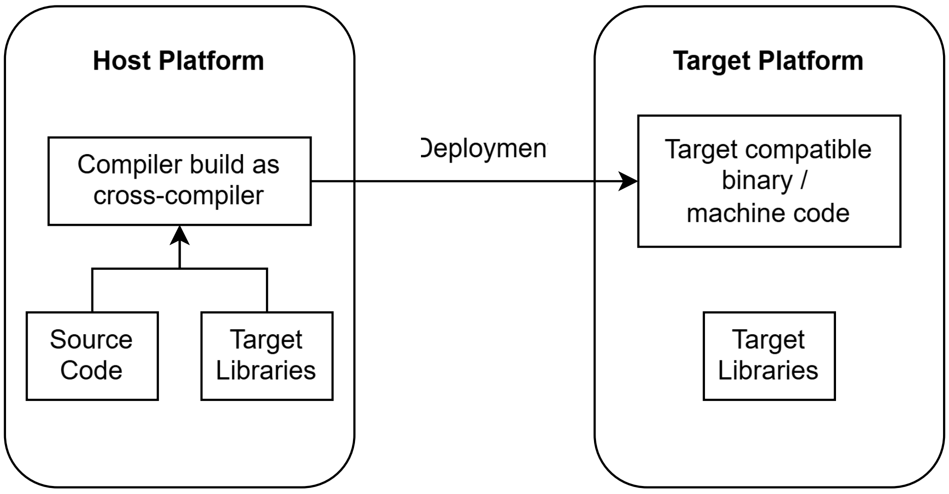


Figure 4.1: Basic diagram of cross-compilation.

to be able to compile code natively. The Linux distribution is also set up with a preinstalled GCC 13. However, this compiler tool-chain not optimal to exploit wholly the MILK-V Pioneer hardware resources. GCC 13 is in fact yet too naive and doesn't feature support for the vector extension version of the hardware, in this case, the 0.7.1. Other than vectorization support GCC 13 doesn't grant access also to several specific CPU optimizations, and finally also several vendor extensions. Thus it was necessary to find and build a tool-chain capable of exploiting the whole potential of the platform.
The manufacturing company of the core C920 design, Xuantie, developed its customized version of the GNU toolchain based on the GCC compiler that fully supported all the extensions and characteristics of its designs. Unfortunately, due to the main usage of the Xuantie and T-Head RISC-V cores in embedded systems, the only pre-compiled versions available were for some reduced number of Linux

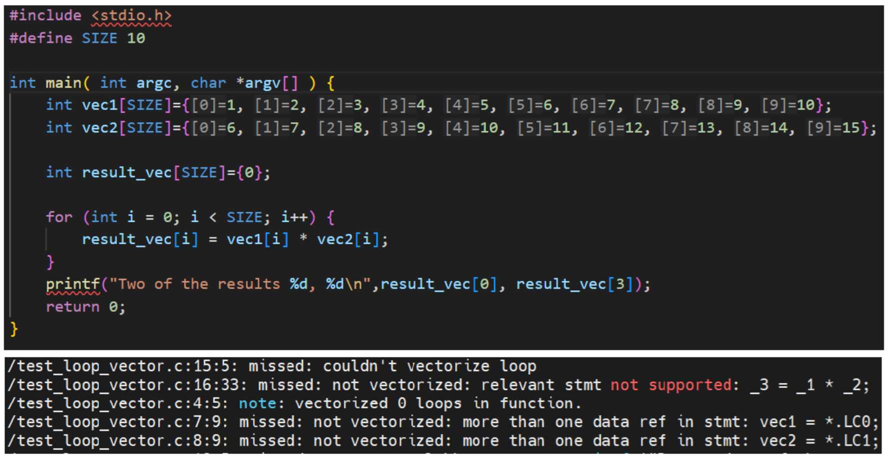
distributions and the x86 platform. Because of this, a non-trivial initial challenge stands in the building and compilation process of a compiler from source code.

Noteworthy, building GCC can generate several errors and issues, spanning from path settings to multi-threading issues during the compilation(e.g. concurrency for the tmp folder between the compiler thread workers). Due to the lack of informative documentation, other problems appeared during the different tries to build this code. This included missing or not finding key libraries and headers like GMP, MPFR or MPC in the system or failing to perform the different staged building, being able to successfully perform compilations and optimizations through stages 1 and 2 but causing errors in the final stage to have the definitive build. After several errors and corrections, and contacting other research teams, it was possible to have a working version of the Xuantie GCC compiler that could natively compile in the MILK-V machine.

Other alternatives that also required a specific native compilation for the machine was a custom version of the LLVM, more specifically the C/C++ language compilers, under development by RuyiSDK [47]. This alternative was based on an older version of LLVM has progressively included different T-head extensions to the compiler to better support this family of platforms. LLVM showcased in other works [48] its superior characteristics and potential performance for RISC-V HPC architectures with respect to GCC. However, in none of their version, the vector extension draft was supported, whereas in older versions of GCC such as 8.x or 10.x provided minimum support for the 0.7.1 vector extension version.

A way to overcome this limitation is by compiling the code into an assembler using the support of version 1.0 and finally parsing the assembler instructions to convert the incompatible instructions and formats (1.0 instructions and structure are not completely retro-compatible with 0.7.1). Therefore, the priority is to use 0.7.1 standard, instructions and C intrinsics, as it is natively supported by the vendor compiler, leaving this more hand-craft and costly proccess of integrating 1.0 into 0.7.1 for cases where it is the only option as an exception.

Auto-vectorization support for RISC-V on either GCC and LLVM is not complete and is usable only for very simple and trivial cases. For instance, GCC can auto-vectorize only data accesses, while for any other operation, the compiler showcases its limitations (example of missing simple vectorization opportunities in Fig. 4.2). LLVM prevails more on this front, but still lacks proper and reliable support for this fundamental part, pushing any optimization tentative to the use of manually written vectorized code.
Regarding the inference framework, in this thesis we employed llama.cpp frozen at commit 5ca0944. This is due to its open-source approach, transparency, and flexibility of being written mostly in C/C++ for both its underlying ggml library and higher-level APIs. After successfully compiling it, llama.cpp provides a main program interface that allows the inference of pre-downloaded models in the



```c
#include <stdio.h>
#define SIZE 10

int main( int argc, char *argv[] ) {
    int vec1[SIZE]={[0]=1, [1]=2, [2]=3, [3]=4, [4]=5, [5]=6, [6]=7, [7]=8, [8]=9, [9]=10};
    int vec2[SIZE]={[0]=6, [1]=7, [2]=8, [3]=9, [4]=10, [5]=11, [6]=12, [7]=13, [8]=14, [9]=15};

    int result_vec[SIZE]={0};

    for (int i = 0; i < SIZE; i++) {
        result_vec[i] = vec1[i] * vec2[i];
    }
    printf("Two of the results %d, %d\n",result_vec[0], result_vec[3]);
    return 0;
}
```

```
/test_loop_vector.c:15:5: missed: couldn't vectorize loop
/test_loop_vector.c:16:33: missed: not vectorized: relevant stmt not supported: _3 = _1 * _2;
/test_loop_vector.c:4:5: note: vectorized 0 loops in function.
/test_loop_vector.c:7:9: missed: not vectorized: more than one data ref in stmt: vec1 = *.LC0;
/test_loop_vector.c:8:9: missed: not vectorized: more than one data ref in stmt: vec2 = *.LC1;
```

Figure 4.2: Example of simple loop addition and assignment missed by auto-vectorization with Xuantie GCC compiler.Top: code, Bottom: GCC compilation information

supported formats, in this case, GUFF. This parameter-based execution inference program allows the user to decide many important aspects of the application such as the number of threads, model file, way of interacting (single run vs interactive continuous execution), number of tokens to predict, and the context size among others. Llama.cpp also provides extra tools to ease the testing of LLMs inference as extra examples for bench-marking inference and matrix multiplication performance.

4.1.2 Effect on core mapping and NUMA regions

CPUs like the Sophon SG2042, are not only multi-core but also are composed of many cores with different memory levels and NUMA regions. Therefore, they face specific problems when trying to parallelize memory-intensive workloads. While trying to utilize all the cores, correctly distributing the workload and the data as a consequence is crucial to achieve good scalability. Thanks to the Linux open tools, we explored the internal configuration of the CPU and how the Operating System (OS) was aware of this division in sub-regions and clusters. Also these tools, despite the limited support for RISC-V 64-bit Linux platforms, illustrated more information about the relative speed of communication between regions and their affinity with cores.

As shown in Fig.4.3, the platform is comprised of 4 NUMA regions, each with an assigned memory section of approximately a quarter of the total RAM, each composed of 16 cores.

Contrary to what expected internal ordering, the cores assignments are not completely contiguous inside of a single region, but rather in blocks of 8 cores, as

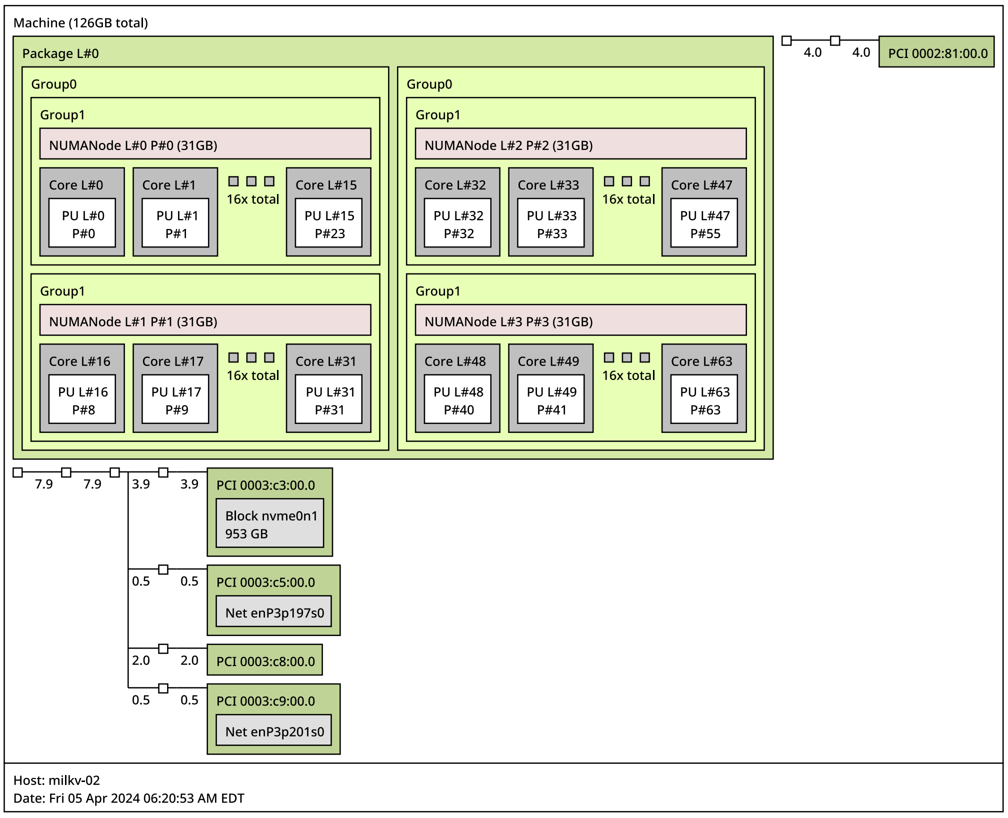
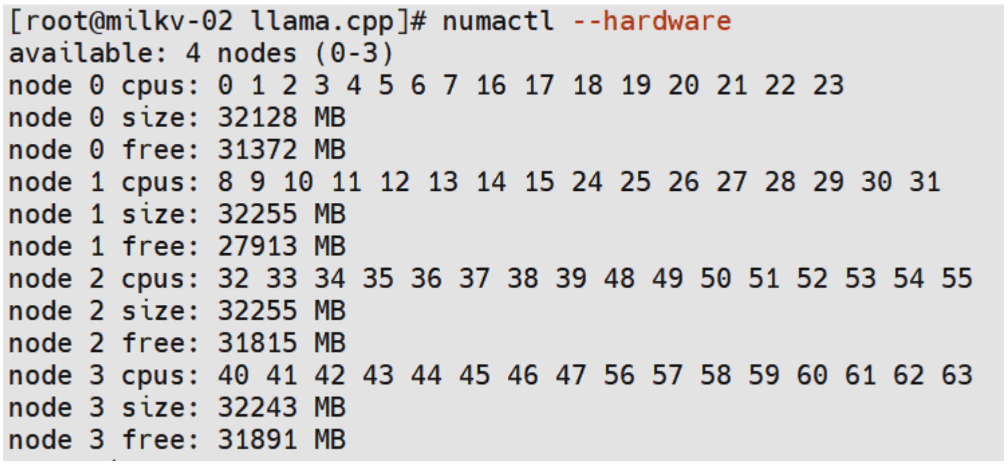


Figure 4.3: Hierarchy and elements of the whole system of the MILK-V Pioneer, obtained with the tool "lscpu" and "lstopo". L# refers to logic ID and PU# Processing Unit ID and P# is the Physical ID

shown in the figure 4.4. For instance, the cores from 0 to 7 and from 16 to 23 integrate the NUMA region 0, while the region 1 is composed of cores from 8 to 15 and from 24 to 31.

Due to the nature of NUMA systems and to better understand the best configuration for multicore workloads, it is important to understand how the regions relate to each other in terms of data exchange speed. Tab.4.1 reflects the different relative latencies between the regions. It is clearly seen that there's more affinity between the groups of regions 0 and 1 on one part and 2 and 3 on the other. This provided a clear map of how different cores relate to each other and should be used for parallelization of the workloads.

There are several ways to exploit this information in general applications, one of these is OpenMP core mapping. However our llama.cpp baseline only supported one customization method for thread allocation and mapping, using the Linux tool numactl. This software offers the user the control to define and establish processes



```
[root@milkv-02 llama.cpp]# numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 32128 MB
node 0 free: 31372 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 32255 MB
node 1 free: 27913 MB
node 2 cpus: 32 33 34 35 36 37 38 39 48 49 50 51 52 53 54 55
node 2 size: 32255 MB
node 2 free: 31815 MB
node 3 cpus: 40 41 42 43 44 45 46 47 56 57 58 59 60 61 62 63
node 3 size: 32243 MB
node 3 free: 31891 MB
```

Figure 4.4: All core Physical IDs of all the 4 NUMA regions (nodes), its assigned RAM memory, and obtained with the "numactl" tool.

NUMA Node	0	1	2	3
0	10	15	25	30
1	15	10	30	25
2	25	30	10	15
3	30	25	15	10

Table 4.1: Numa Node/Region latency relative to each other, obtained using numactl tool

NUMA scheduling or memory placement policies. These policies affect not only the main program but also all of its children.

The capability of forcing a certain NUMA policy overwriting the default behavior of Linux OS. This conventional behavior is typically based on the strategy of NUMA balancing, which is convenient for consumer usage when multiple applications run on a single machine. But this same policy can hinder in great measure high-performance applications that run standalone consuming most of the machine resources [49].

Numactl in combination with the execution of llama.cpp allowed to provide some simple directives about direct core-thread allocation mapping or memory allocation strategies such as interleaving with round-robin across NUMA regions. Both of these alternatives (the core/memory binding, matching NUMA regions and adapting to the number of threads, and the interleave strategy) have been evaluated and compared, also in combination with the disabling or the NUMA balancing default policy.

Another important evaluation during this exploration was the usage of the “perf” tool, to better understand the effect of a certain NUMA policy and its impact on memory usage. This profiling tool allows also us to evaluate the impact of the optimizations accomplished by the different compilers. "Perf" uses near-hardware level counters, tracepoints, and events to provide key metrics such as cycles of execution, number of instructions, and page/cache misses at different levels and is a key tool in computer science for benchmarking the performance of programs as shown in the Fig.4.5.

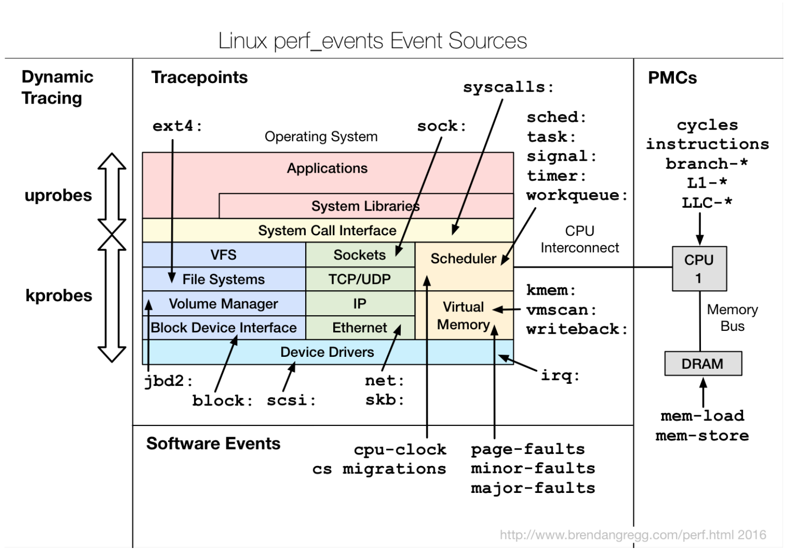


Figure 4.5: Diagram with different hardware metrics, counter, events and tracepoints that the tool "perf" can measure.

4.2 LLMs inference analysis

Following, we'll dive deep into llama.cpp details and its execution process step-by-step. This process allows to achieve improvements at the framework level that could better fit the underlying hardware micro-architecture. For instance, for the Xuantie CPU by exploiting the vector unit. This require understanding how the inference process is carried out by llama.cpp and the usage of different functions in the ggml back-end library.

4.2.1 Inference Bottlenecks

The inference process of a modern LLM incurs a lot of compute and memory-intensive operations. The main target of this optimization is the attention mechanism as experiments, such as the one of Fig.4.6, showcase that most of the computational time is spent during the attention blocks. One of the main prob-

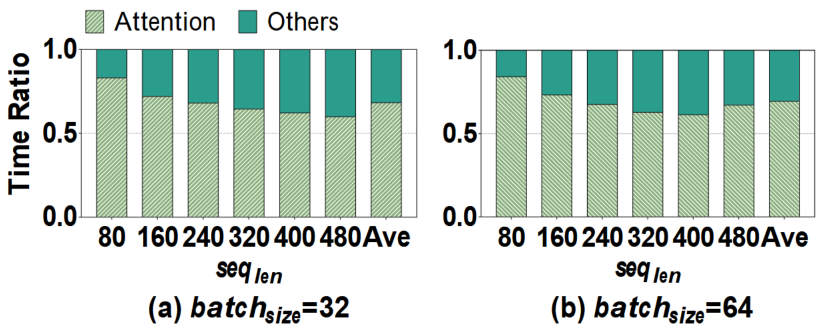


Figure 4.6: Time breakdown of the Bert-base model on a 64core ThunderX2 server. Ran using PyTorch 2.0.0 with BLIS as the backend. Extracted from [50].

lems is the bottleneck caused by memory accesses, as the computational speed is higher than memory in memory-bound problems. Memory-bound problems are characterized by their low Arithmetic intensity in which the loaded data are used in a few operations, while the cost of bringing them to the CPU registers from most probably external DRAM or in better cases, internal cache memories, is not compensated. Following this, it's possible to evaluate the time spent during each inference operation. In fact, llama.cpp allows for more fine-grained profiling through an additional compile-time flag. This is achieved by including counters for each operation and activating them through compile-time directives. Finally, the results are shown, not only after each inference run but after the processing of each token generation, showcasing a full execution for generating a token.

The analysis performed reveals that, as expected, the operation that incurs in most of the computational time is the matrix multiplication, as shown in Tab.5.1. In fact, this operation takes approximately 91% of the total inference time. This is completely in line with the theoretical response and highlights the importance of having efficient data movement and optimized implementations that can improve this operation.

4.2.2 Matrix multiplication: BLAS libraries and vectorized kernels

Since GEMM operations are fundamental, it is key to observe the performance of different implementations. To evaluate a pure matrix behaviour a custom test environment was developed to simplify the analysis and have better control over the test-bench and implied variables.

Our test environment is a custom variation of a public matrix-multiplication repository used for Cornell University subject CS 5220 [51]. This repository was modified by improving the flexibility of the tested matrix multiplications by having more tunable dimensions to increase the dimensions capabilities. Furthermore, the corresponding Makefile to include the Xuantie GCC compiler build in the system, add extra configurable flags, and even have the possibility of executing it using the OpenBLAS options, shown in listing 4.1. All of these options allowed us to automate the testing task through bash scripts.

Listing 4.1: Makefile used to include the Xuantie GCC and the required optimizations for the C920 core

```
BUILDS= intr_m1 intr_m2 intr_m4 intr_m8

# C and Fortran compilers
CC=/scratch/tools/compilers/xuantie_gcc/bin/riscv64-unknown-linux-gnu-gcc
FC=gfortran
LD=/scratch/tools/compilers/xuantie_gcc/bin/riscv64-unknown-linux-gnu-gcc

# Compiler optimization flags.
TARGET= -mcpu=c920
OPTFLAGS= -O3 -ftree-vectorize -fopt-info-vec-all -ftree-slp-vectorize

C_INC_PATHS= -I/scratch/tools/compilers/xuantie_gcc/lib/gcc/riscv64-unknown-linux-gnu/10.4.0/include -I/scratch/tools/compilers/xuantie_gcc/lib/gcc/riscv64-unknown-linux-gnu/10.4.0/include-fixed
CFLAGS=$(TARGET) $(C_INC_PATHS)

LD_PATHS= -L/scratch/tools/compilers/xuantie_gcc/lib/ -L/scratch/tools/compilers/xuantie_gcc/lib64/
LD_FLAGS= $(TARGET) $(LD_PATHS)

# Add -DDEBUG_RUN to CPPFLAGS to cut down on the cases.
CPPFLAGS= "-DCOMPILER=\"$(CC)\"" "-DTARGET=\"$(TARGET)\"" "-DC_INC_PATHS=\"$(C_INC_PATHS)\"" "-DOPTFLAGS=\"$(OPTFLAGS)\""

# Compile a C version:
LIBS = -lm -lrt
OBJS = matmul.o

# Libraries and include files for BLAS
LIBBLAS=-lopenblas
INCBLAS=-I/scratch/jpoveda/simple_vectorization/OpenBlas_910V_No_Fortran/include/
```

Considering this, adding other implementations that could generate great results in the platform has been the goal. Despite the lack of any specific work that used the C920 and BLAS libraries directly, other works targeting also related hardware, such as the Xuantie C910 or the C906, used OpenBLAS as the BLAS library [52]. OpenBLAS specifically supports Level 1,2,3 BLAS implementations with customized vector kernels for the C910 hardware, which makes it a perfect candidate to be used with our hardware.

On the other side, it was interesting to compare it with how the vector unit would work with different Vector register grouping (LMUL) values. Despite being more flexible as vector register assignment is simpler, using small or non-grouping could lead to slower performances and a reduced memory bandwidth [53] according to other experiments and the underlying hardware micro-architecture. To achieve a comparison between the GEMM (General Matrix Multlication) implementation of OpenBLAS and the potential performance with the usage of the vector unit we developed a single precision version of SGEMM with vector C intrinsics as depicted in the listing 4.2.

Listing 4.2: Basic vectorized version of sgemm kernel, using vector grouping of 4 , using RVV 0.7.1 C intrinsics, prerforming operation on one row and one column

```
void sgemm_vec(size_t size_m, size_t size_n, size_t size_k,
               const float *a, // m * k matrix
               size_t lda,
               const float *b, // k * n matrix
               size_t ldb,
               float *c, // m * n matrix
               size_t ldc) {
  size_t vl;
  for (size_t m = 0; m < size_m; ++m) {
    const float *b_n_ptr = b;
    float *c_n_ptr = c;
    for (size_t c_n_count = size_n; c_n_count; c_n_count -= vl) {
      vl = vsetvl_e32m4(c_n_count );
      const float *a_k_ptr = a;
      const float *b_k_ptr = b_n_ptr;
      vfloat32m4_t acc = vle32_v_f32m4(c_n_ptr, vl);
      for (size_t k = 0; k < size_k; ++k) {
        vfloat32m4_t b_n_data = vle32_v_f32m4(b_k_ptr, vl);
        acc = vfmacc_vf_f32m4(acc, *a_k_ptr, b_n_data, vl);
        b_k_ptr += ldb;
        a_k_ptr++;
      }
      vse32_v_f32m4(c_n_ptr, acc, vl);
      c_n_ptr += vl;
      b_n_ptr += vl;
    }
    a += lda;
    c += ldc;
    }
}
```

4.2.3 Kernel development and integration

Having analyzed matrix multiplication implementations, the following part is the integration into llama.cpp. The first approach tried to integrate OpenBLAS as the computing backend of llama.cpp, an option minimally affected and supported by the repository. After successfully building it, it was noticed that the performance during inference was minimally affected by this change. According to the previous test, this didn't correspond to the expected behavior, as OpenBLAS and its specific implementations is superior to any scalar performance. The reason for this can be found inside of the code corresponding to the ggml implementation of the function "ggml_compute_forward_mul_mat_use_blas(struct ggml_tensor * dst)" (listing 4.3).

Listing 4.3: ggml.c in function ggml_compute_forward_mul_mat_use_blas() with added comments in the execution of a quantized Q4 model for clarity

```
    // NOTE: with GGML_OP_MUL_MAT_ID we don't want to go through the BLAS branch because it will dequantize (to_float)
    // all the experts for each batch element and the processing would become incredibly slow
    // TODO: find the optimal values for these
    if (dst->op != GGML_OP_MUL_MAT_ID && //kind of operation to be performed
        ggml_is_contiguous(src0) && //matrix 1 typical type is Q4_0 if model is Q4_X quantized
        ggml_is_contiguous(src1) && //matrix 2 typical type is FP32
      //src0->type == GGML_TYPE_F32 &&
        src1->type == GGML_TYPE_F32 && //datatype of matrix 2
        (ne0 >= 32 && ne1 >= 32 && ne10 >= 32)) { //dimensions of matrix in & out

        return true;
    }
```

This function is in charge of deciding if llama.cpp has been built with any alternative BLAS backend, whether it is convenient and worth using the external BLAS implementation with selection criteria based on the dimensions of the matrices if the source matrices are contiguous in memory, the data type of the operand and heuristics. Whenever these conditions are not met, llama.cpp will continue with its default strategy and default ggml implementations.

To better understand how each criterion affects the usage of OpenBLAS, we expanded llama.cpp with more profiling. The additional profiling data revealed that the dimensionality criteria was the most common issue. Further exploration into the ggml approach for matrix multiplication and internal operations was required. For this reason, we developed a Python script to parse all the data that we outputted to a text file on every matrix multiplication operation thanks to further modifications in the llama.cpp source code. This allowed an in depth analisys even of the dimensions used during inference and in the whole inference as is seen in the Tab. 4.2.

Dimensions (shared x dst_0 x dst_1)	Count	Percentage
'4096 x 4096 x 1'	11904	41.6%
'4096 x 11008 x 1'	5964	20.8%
'11008 x 4096 x 1'	2982	10.4%
'32 x 128 x 1'	2496	8.7%
'128 x 32 x 1'	2496	8.7%
'64 x 128 x 1'	480	1.7%
'128 x 64 x 1'	480	1.7%
'4096 x 4096 x 6'	384	1.3%
'4096 x 4096 x 2'	384	1.3%
'4096 x 11008 x 6'	186	0.7%
'4096 x 11008 x 2'	186	0.7%
'4096 x 32000 x 1'	99	0.3%
'32 x 128 x 6'	96	0.3%
'32 x 128 x 2'	96	0.3%
'128 x 32 x 6'	96	0.3%
'128 x 32 x 2'	96	0.3%
'11008 x 4096 x 6'	93	0.3%
'11008 x 4096 x 2'	93	0.3%

Table 4.2: Example of result of matrix multiplication during inference using llama-2-7b.Q4_0, with the prompt "Once upon a time", generating 20 tokens.

The results showed that most matrix multiplications performed during an LLM execution are performed mainly as matrix-vector multiplications. The weights are stored as a matrix while the other operand is instead a tall-and-skinny matrix(TSMM) usually. Therefore, as the criteria already pointed out, smaller matrixes can greatly affect the performance of BLAS-like approaches to matrix multiplication.

This problem, defined as the Skinny and Tall matrix multiplication [54], related

to non-regular dimensions for matrices requires a different approach than regular BLAS tiling and micro-kernel approach, as the typical size of these kernels requires a minimum dimension of 8x8 or 4x16 to fully utilize the personalised kernels and memory awareness loading.
Llama.cpp and ggml process of choosing a matrix implementation and execution flow depends on the hardware support, the data types, and the dimensions. Furthermore, the data types during execution are not the same, as OpenBLAS and other BLAS libraries support most commonly only single, and double precision floating point arithmetic, while the default implementation can work using integer arithmetic and after that use conversion on the result, as shown in algorithm 1.

Algorithm 1 Psedo-code algorithm for llama.cpp, ggml quantized matrix multiplication decision making with BLAS build

```
 1: if Source tensors are contiguous & Dimensionality is correct then
 2:     Dequantize input tensors
 3:     Cast to Float
 4:     Perform fp32 BLAS matrix multiplication        ▷ Result is already in float
 5:
 6: else if Supported Platform and Quantization type  then
 7:     Execute custom matrix-matrix multiplication
 8:     ▷ Q4 and fp32 matrixes (fp32 is quantized to Q8), widening integer operations, final cast to fp32
 9:
10: else                                    ▷ Default quantized matrix multiplication
11:     Division in chunks on the input
12:     Quantization and operation in integer data   ▷ The data type increases to maintain precision
13:     Cast to float and storage of result
14: end if
15:
```

This behaviour makes BLAS libraries have to make more compute-intensive casting operations as weight tensors are bigger square matrices, while the output is usually a vector as we have seen previously. Therefore the casting into float operation is order of magnitude higher for BLAS workflow (e.g. for LLAMA 2 and its design dimensions the embedding and weights have dimension 4096x4096 vs a single result vector row of 4096 values).

As a result of this analysis and the overhead brought by the use of operations in floating point precision, our proposal stands in the definition of a kernel targeting one of the most popular quantization techniques, Q4_0. Noteworthy, this proposal can be further expanded to support each data type to obtain even more throughput. We tackled Q4_0 quantization due to its balance between model performance/behaviour and model size, as well as its simpler storage scheme. The latter is based on blocking and scaling. Q4_0 quantization utilizes blocks of 32 weights, coupling 2 int4 values in 1 int8 that has to be divided during the dequantization process. These values share a scale factor stored in a half-precision floating point and the quantization follows the following formula:

$$weight(fp32) = weight(int4) \times scalefactor(fp16)$$

Dequantization formula for Q4_0 quantization and storage datatypes

Thus, model data is not only the weight themselves but also their scale factor, all stored in the same blocks. Current implementations for specific hardware, due to the template common to all of them, only allow processing one weight block at a time, therefore having very little room for optimizations except the usage of vectorized kernels and specific instructions.

Initially, to get a direct vectorized baseline with llama.cpp default computing flow, a simple kernel doing basic operations(unpacking int4 into int8, and multiply and add operations) has been developed. This implementation has been placed inside the *ggml_vec_dot_q4_0_q8_0* function, which gets used by llama.cpp to perform vector multiplication between a Q4_0 vector and a Q8_0 one. The kernel has 4 different sections that can be also obseverd in the code snippet 4.4:

- **Loading**: It is loaded with one set of 16 int8 values that contain the Q4 values and then two blocks of 16 to have a total of 32 Q8 values.

- **Split and reinterpret**: There are used masks to obtain the upper lower Q4 and shifting for the upper Q4. Later this unsigned values are modified to be reinterpreted as int8 operands even if they are only int4.

- **Calculation**: We use half of the Q8 (y0) to multiply against one of the Q4 vector(v0) while widening the operation to int16 to maintain precision. After that, the result is accumulated with the result of the multiplication among the other half of the Q8 (y1) and the other Q4 vector (v1). The final result is a vector of 16 elements in int16. These values have to be added to each other. Therefore is used the reduction by addition using a zero vector, resulting in a final result in int32 as this operation is also widened.

- **Scale**: The result of the multiplication is casted to a single precision floating point, as well as the scale factors of x and y and they are all multiplied and accumulated to whay would be at the end of the loop iteration a final result.

Listing 4.4: function ggml_vec_dot_q4_0_q8_0() with vectorized implementation with added comments, in ggml_quants.c

```
void ggml_vec_dot_q4_0_q8_0(...){
  ...
  const int qk = QK8_0; \\ 32
  const int nb = n / qk; \\Number of blocks to process
  ...
  const block_q4_0 * restrict x = vx;
  const block_q8_0 * restrict y = vy;
  ...
  float sumf = 0.0;
  //first impl -> currently NB must be even
  size_t vl = vsetvl_e8m1(qk/2);
  assert(vl == qk/2);
  for (int i = 0; i < nb; i++) {
  // LOADING
    // load 32 Q4 elements in 16 int8
    vuint8m1_t tx = vle8_v_u8m1(x[i].qs, vl);
    //load 32 Q8 elements in 2 16 variables
    vint8m1_t y0 = vle8_v_i8m1(y[i].qs, vl);
    vint8m1_t y1 = vle8_v_i8m1(y[i].qs+vl, vl);
   //SPLIT AND REINTERPRET
    // Extract Q4 with lower mask and shifting
    vuint8m1_t x_a = vand_vx_u8m1(tx, 0x0F, vl);
    vuint8m1_t x_l = vsrl_vx_u8m1(tx, 0x04, vl);
    vint8m1_t x_ai = vreinterpret_v_u8m1_i8m1(x_a);
    vint8m1_t x_li = vreinterpret_v_u8m1_i8m1(x_l);
    vint8m1_t v0 = vsub_vx_i8m1(x_ai, 8, vl);
    vint8m1_t v1 = vsub_vx_i8m1(x_li, 8, vl);
    //CALCULATION
     //First we multiply widening to int16 and obtain results
    vint16m2_t vec_mul1 = vwmul_vv_i16m2(v0, y0, vl);
     //Then we multiply accuulate widening to int32 to obtain results
     vint16m2_t sum_vector = vwmacc_vv_i16m2(vec_mul1,v1, y1, vl);
     vint32m1_t vec_zero = vmv_v_x_i32m1(0, vl);
     vint32m1_t mask = vmv_v_x_i32m1(0xffffffff,vl);
     //We reduce the final vector of results adding all of them
     vint32m1_t vs4 = vwredsum_vs_i16m2_i32m1(mask, sum_vector,
   vec_zero, vl);
    //We extract the result
     int sumi = vmv_x_s_i32m1_i32(vs4);
    //SCALE
     //We multiply by the scaling factors of x and y, and acumulate
     sumf += sumi*GGML_FP16_TO_FP32(x[i].d)*GGML_FP16_TO_FP32(y[i].d)
   ;
    }
    *s = sumf;    //We save the result
```

With the baseline kernel version, we tried to continue with this methodology by developing more refined kernels, although still wrapped into the default llama.cpp flow. More in detail, this meant that no additional change to higher-level chunking and multi-threading policies have been modified. However, these tentative kernels, added levels of complexity to the kernel that impacted negatively the performance, which seemed counter-intuitive. The likely cause of this lack of performance enhancement has been attributed to llama.cpp default chunking, which doesn't allow for bigger improvement margins.

This, and other limiting factors, such as the vector-vector usage instead of a matrix-vector or matrix-matrix one, encouraged us to develop a more complete solution that is used instead of the default approach if certain criteria are met. For instance, our proposed approach focuses, as previously stated, on Q4_0 versus Q8_0 tensors. Other criteria refer to the even dimensionalities and the contiguous memory storage of the tensors.

Our implementation aims to improve data re-usage and register utilization, by unrolling the kernel over 2 rows of the matrix. Another optimization that increases data re-usage, is the use of 2 data blocks at a time in each row. By moving through the whole row dimension of the weights 2 results in single-precision floating point are generated as shown in Fig.4.7.

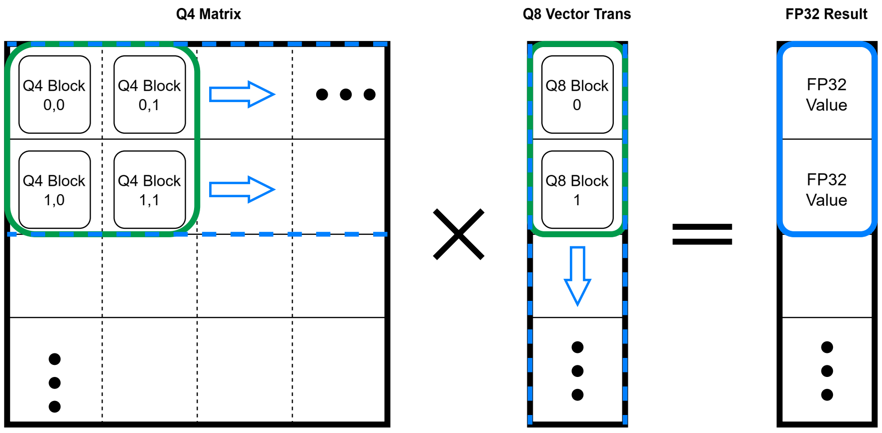


Figure 4.7: Diagram of kernel strategy where green elements are multiplied and accumulated in the same most inner loop and we move through the rows to generate 2 results (blue colour) with Q4 matrix and Q8 vector (or column of matrix)

The strategy used reduces the loading of the Q8 data in half due to the loop unrolling and increases the arithmetic intensity of the operation, increasing significantly the utilization of vector registers compared to the RVV 0.7.1 kernel. However, due to less efficiency obtained with the Xuantie Compiler, we used the newer GCC 14.2 with the *xtheadvector* extension to generate an assembly function that could be embedded into the ggml framework for the matrix multiplication.

Further analysis of the kernel compilation with the C vector intrinsics, in the first iterations, revealed that the Xuantie GCC compiler for C, even compiling the kernel as an external function, did not use all the immediate functions available, doing extra loadings, and performing extra crrs instructions that with simpler, less unrolling, even if the same structure was used, generating extra instructions and being less optimal. This showed a successful result but probably due to the extra loading and storage of the intermediate floating-point results, the performance was slightly less than expected.

Therefore, a more complex version integrated the most-inner look through the rows and the storage of the final 2 results of each iteration into a more efficient assembly function. In fact, the latter is more aware of the whole picture for greater optimization. The usage of each version could be toggled with an external flag "Velorisk_complex", seen in code fragment 4.5, as well as a general flag to use our kernels or the default behaviour.

Listing 4.5: function ggml_vec_dot_q4_0_q8_0() with vectorized implementation with added comments, in ggml.c

```c
for(int k=0 ; k<ne1 ; k++ ){ //Number of columns output
   for(int h=0 ; h<ne0 ; h+=2 ) //number of rows matrix, and we
    unroll the matrix 2 times
   {
     const block_q8_0 * restrict y = ((const block_q8_0*)wdata) + k *
    n_blocks_q8;
     const block_q4_0 * restrict x_0 = ((const block_q4_0*)src0->data)
     + h * n_blocks;
     const block_q4_0 * restrict x_1 = ((const block_q4_0*)src0->data)
     + (h+1) * n_blocks;
#if defined(VELORISK_COMPLEX)
     kernel_complex(ne01,ne0, x_0,x_1, y,dst_temp, k, h, n_blocks );
#else
     float sumf_0 = 0.0;     //Temporary result
     float sumf_1 = 0.0;     //Temporary result
     for (int i = 0; i < n_blocks; i+=2)    //number of blocks to
    process, unroll2 2 times
         kernel( ... );
     *(dst_temp+h+k*ne0) = sumf_0;         //Storage of results
     *(dst_temp+(h+1)+k*ne0) = sumf_1;     //Storage of resuls
#endif
         }
     }
```

The next natural step was integrating multi-threading into our kernel. Llama.cpp uses an internal variable that variates depending on the thread that it corresponds to called *ith* and stores the total number of threads in an execution in the variable *nth*. This variable allows us to parallelize our refined implementation in most inner loop if we exclude the one included in this version of the kernel across the unrolling of the rows of the matrix as displayed in 4.6.

Listing 4.6: Multithreaded loop modification with added comments in ggml.c

```
for(int k=0 ; k<ne1 ; k++ ){ //Number of columns output
        for(int h=ith*2 ; h<ne0 ; h+=2*nth ) //number of rows and we
    unroll the matrix 2 times
        {
        ...
        }

```

Using these integrated global variables, allows us to include in our kernel up to the number of thread defined by the user, without the requirement of PRAGMAS or any external library that is not already used by llama.cpp.

Chapter 5

Results

The following chapter will illustrate and provide an analysis of the most relevant results obtained during the experiments carried during this thesis. The RISC-V HPC platform used, as explained previously, is the MILK-V PIONEER one, with the CPU Sophon SG2042 and a configuration of 128GB of DRAM in a single stick. This machine is running Fedora Linux 38 (Workstation Edition) with a kernel version of Linux 6.1.13 for the riscv64 architecture. For the inferece, we used the publicly available repository of llama.cpp frozen at the commit 5ca0944a153b65724d51b2f484139aa25ccb7a8b, with date of 4 of June of 2024.

5.1 Inference bottleneck analysis

Using llama.cpp as the inference framework, allowed performing an internal analysis of the underlying operations and the time used during inference. When running an inference with this build configuration, llama.cpp will rely on internal counters to output the timing of each operation. An example of this is showcased in Tab. 5.1, where in the left column there are defined the main types of operations used during inference and their corresponding time.

These results confirmed one of the main bottlenecks in our machine, the matrix multiplication, where it usually takes more than 90% of the time during the inference execution.

Additionally to this profiling, we also explored how the different vector grouping affected the matrix multiplication performance in this machine. Also, we analyze how its performance could compare to OpenBLAS when using the custom matrix-multiplication test-bench described in sub-chapter 4.2.2. The results in Fig.5.1 report the importance of choosing the correct register grouping and the superiority that OpenBLAS has with its GOTOBLAS Algorithm, multi-level tiling, and specialised kernels.

Perf ggml operation	**Total time (ms)**
ADD	5.07
MIL	3.66
RMS_NORM	2.51
MUL_MAT	297.048
CPY	2.755
CONT	0.508
RESHAPE	0.279
VIEW	0.55
PERMUTE	0.271
TRANSPOSE	0.139
GET_ROWS	0.045
SOFT_MAX	3.208
ROPE	11.183
UNARY	7.446

Table 5.1: LLama.cpp default timing results for the one token generation with the model Llama-2-7B-GGUFllama-2-7b.Q4_K_M.gguf, with 64 threads and interleave NUMA control policy on MILK-V Pioneer

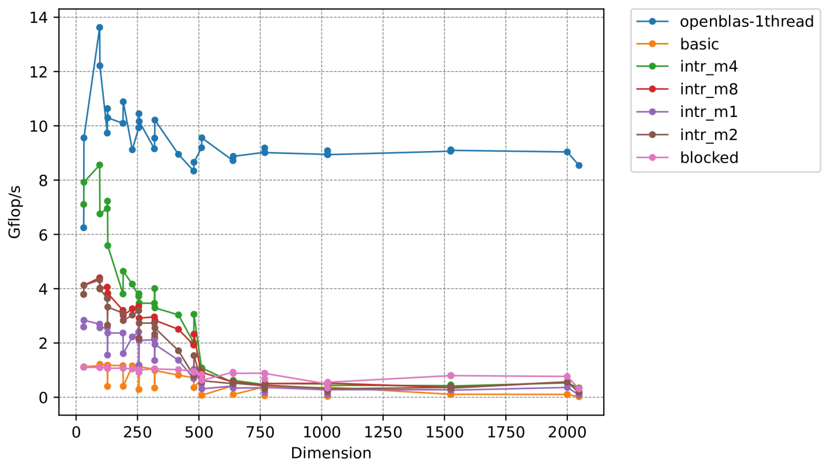


Figure 5.1: Square Matrix multiplication benchmark single core results, variating LMUL, a blocked strategy of 16x16 and OpenBLAS with 910V

As part of our test-bench, we also evaluated the scalability of the most powerful matrix multiplication with the most powerful algorithm for fp32, OpenBLAS, and the effect of the NUMA thread migration policy that is enabled by default in our Linux and in general. The results in Fig.5.2 illustrate how the algorithm scales efficiently with multi-threading and the impact that thread migration generates when using an increased number of threads.

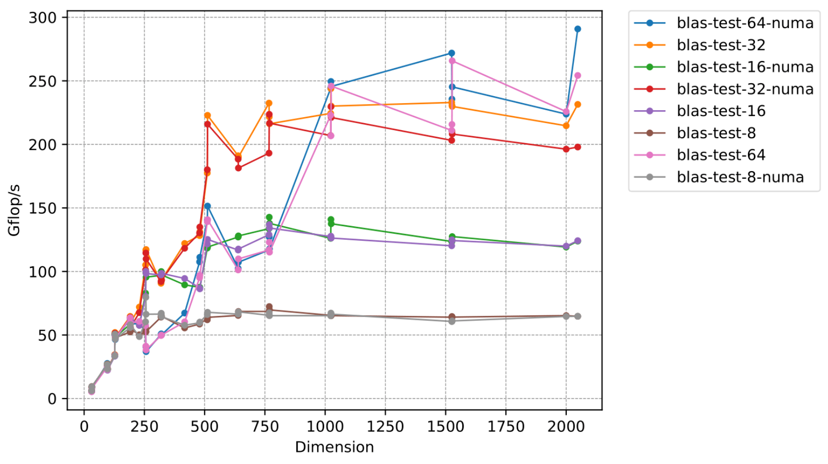


Figure 5.2: SGEMM scalability performance with OpenBLAS with 910V configuration evaluate with and without NUMA thread migration enabled (-numa refers to disabled)

Finally, to better understand how the matrix multiplication operation on the llama.cpp inference framework bottleneck worked, we ran 2 different state-of-the-art LLMs LLAMA 2 7B and Microsoft Phi-3 mini both Q4 and with a short prompt with 5 tokens and a longer one with 69 tokens. The results of the dimension analysis showcased that in 94% of matrix multiplications during inference the output had one dimension 1, thus being effectively a vector-matrix multiplication, as expected in auto-regresive inference, and the rest of the cases were 3% implied a matrix output with dimensionality 2 for the rows (very close to a vector-matrix operation) and final 3% is related with the input tokens number.

This completely matches the analysis of the criteria during BLAS decision usage, as was approximately 3/4 for LLAMA 2 and 2/3 for Phi-3 of the times exclusively because of dimensionality problems, as the second operand is too small, usually a vector or a matrix with 2 rows and the minumum established by the heuristics criteria is 32 rows to use BLAS implementations in matrix multiplication. While, the rest was due to dimensionality and contiguity (tensors not stored sequentially in memory) problems. The contiguity problem can be due to the llama.cpp strategy, in which, when a tensor is required to be transposed, only *nb* and **ne** variables of the tensor are changed but the data itself is not moved, which can generate mismatches from the actual layout in memory and the characteristics of the tensor.

5.2 Micro-benchmarks results

After evaluating the performance in external matrix multiplication, we focused on the quantized models, in particular Q4_0 as it reduces the memory footprint of the model and can offer a balance between precision and size, and is fully integrated into the llama.cpp framework.

Our first test related to inference regarded the efficiency of the different alternatives when performing the Q4_0_Q8_0 matrix multiplication. This experiment was run using a modified version of the *benchmat-matmul.c* example included in the llama.cpp repository and the Xuantie GCC compiler with the configuration optimization for the C920 core and maximum level of optimization (-O3), running in a single-thread environment.

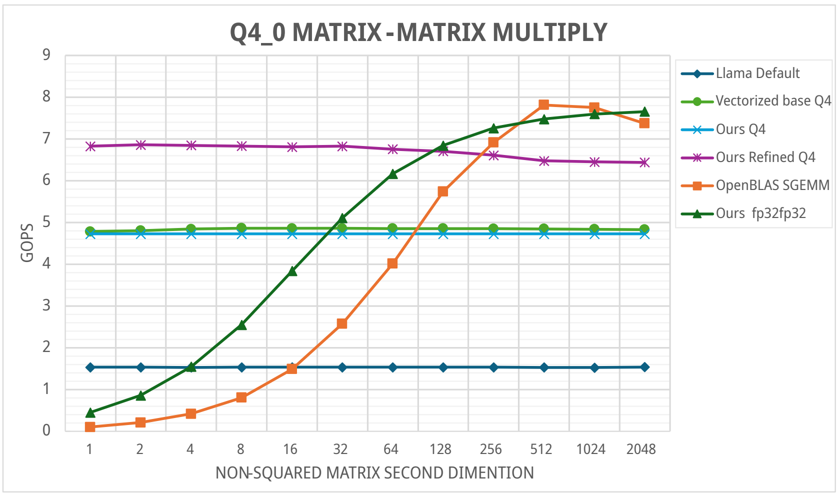


Figure 5.3: Square matrix vs non-regular matrix multiplication benchmark single core results, varying the non-regular dimension, with Xuantie GCC 10.4 compiler

As reflected in Fig.5.3, our more refined implementation is capable of outperforming all the existing options. It is on average 4.36 times faster than the default llama.cpp implementation and gathers a 38% speed-up with respect to the vectorized baseline we developed. Noteworthy, OpenBLAS and the other fp32 version performances when using smaller dimensions are dramatically low. While with bigger tensors they can outperform our proposed implementation. To get these last results and analysis we bypassed the criteria. checking when llama.cpp is built with OpenBLAS in order to have a full execution with this library, if not, it would use the default kernel.

The lower performance can be caused by the dequantization of the Q4_0 weights matrix. As shown in Tab.5.2 the weights dimensions are the same and the dequantization will consume a constant time.

Types of matrixes	**Q4xfp32**	**Q8xfp32**	**fp32xfp32**
Performance (GOPS)	5.52	5.72	5.95
Time (ms)	778.2	750.2	723.1

Table 5.2: Performance of OpenBLAS in matrix-matrix multiplication (4096X4096 X 4096X128) with different data types inside of benchmark_matmul.c

According to these results, in the case of a matrix 4096x4096 the time that it adds would be around 55ms, which will be constant. To put this into perspective, the time it takes to perform a matrix-vector multiplication with baseline vectorized kernel with dimensions 4096x4096 x 4096x1 is approximately 7ms to accomplish a performance of 7.87GOPS. This element added to the poor performance of OpenBLAS SGEMM for tall and skinny matrixes makes it a non-viable option for operations with more "regular" dimensions.

This problem is also evident when we observe the results in matrix-vector multiplication as can be seen in Fig.5.4. In this figure, we benchmarked the performance of the alternatives in a square matrix vs vector multiplication where we variate the shared dimension.

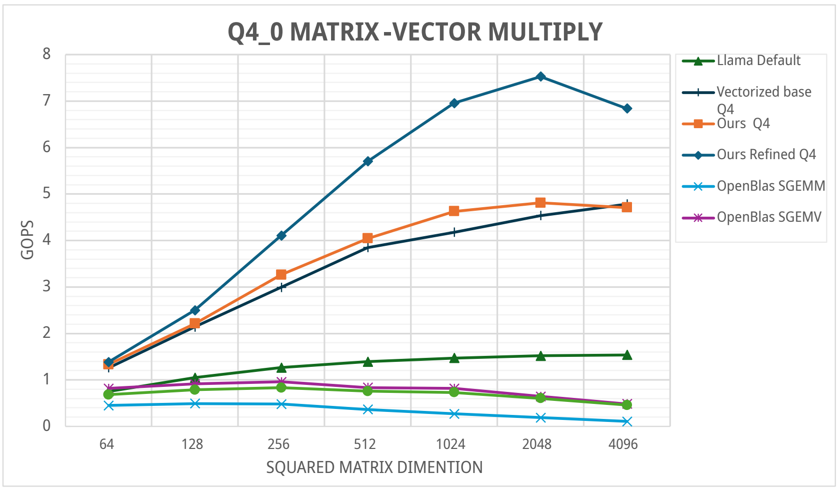


Figure 5.4: Square matrix vs vector multiplication benchmark single core results, varying the vector the dimension, with Xuantie GCC 10.4 compiler

In order to make a fair comparison we used the BLAS level 2 routine (GEMV) that is specifically designed for this kind of operation. Despite being noticeable the performance improvement by the usage of GEMV routine compared to SGEMM OpenBLAS both are really far from the performance shown by our implementation. The best-performing kernel is our implementation version for Q4_0 for each case tested for matrix-vector with a peek performance of 7.5GOPS with the dimensions (2048x2048 x 2048x1).

5.3 Inference results

After some exploratory work and related work, we focused on the impact that this information and our knowledge about the different bottlenecks and expected performance from the different alternatives could have.

5.3.1 Single Core

The boosted in performance showcased by is also observed when running inference with our modified versions on the llama-2 model with 7 billion parameters with a Q4_0 quantization in Fig. 5.6 and Fig. 5.5. The performance reaches up to 4 times the performance for token generation and 4.7 times for prompt processing improvement compared to the default performance. These improvements are only accomplished with our most refined implementation. Whereas, in comparison, the vectorized baseline or the simple version behave similarly. In fact, the vectorized version has an edge of around 2.9% in prompt processing while our simple implementation performs token generation a 6%.

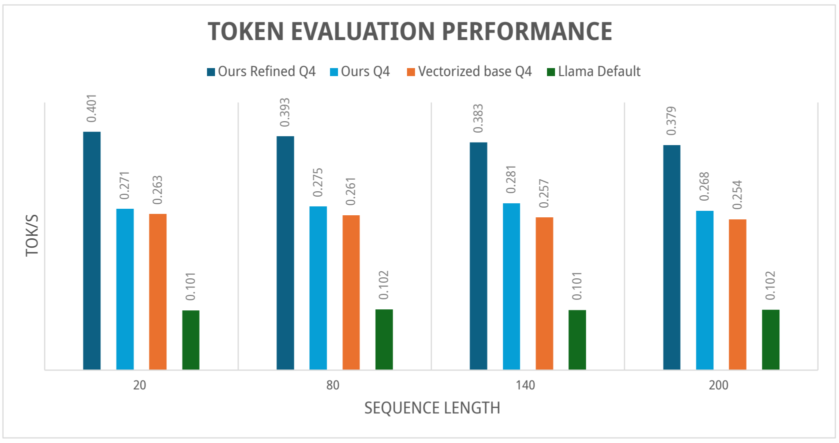


Figure 5.5: Token generation performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" single core results, varying the sequence length generated, compiled with Xuantie GCC 10.4

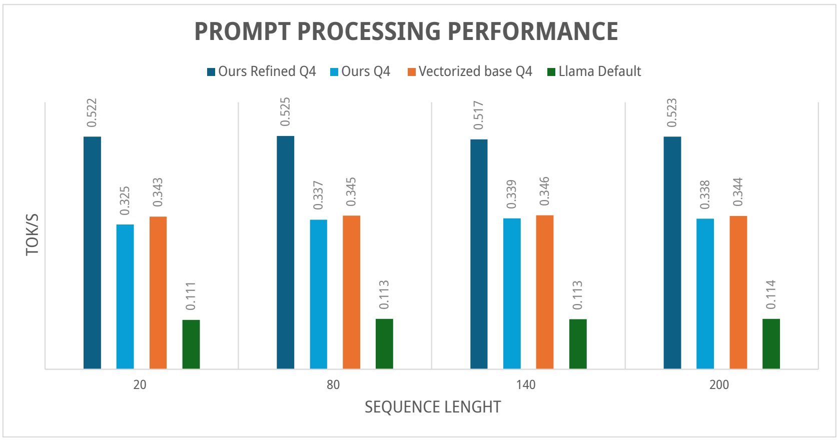


Figure 5.6: Prompt evaluation performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" single core results, varying the vector the dimension, compiled with Xuantie GCC 10.4

The performance accomplished by OpenBLAS, by also bypassing all the minimum size checks, so the execution reveled to be in the range of 0.009 tok/s (one order of magnitude less than default) for token evaluation and 0.08 for prompt processing, making clear the necessity of limiting the usage of OpenBLAS to when the conditions allow to extract the most of its complexity.

5.3.2 Multi-threaded performance

Scalability, compilers and NUMA

After the single-core performance and testing, which demonstrated the capabilities of our implementation, the next test involved scalability by increasing the number of threads and comparing the level of performance that the different compilers built to do native compilation in our hardware. Due to the core limit, and that they don't possess the ability to run more than one thread per core (as in contrast to Intel's Hyperthreading), we escalate from 1 to 64 threads doubling it each time.

Also the available compilers are the default GCC 13.2 that is preinstalled with our Linux distribution, the Xuantie toolchain that includes the customized GCC 10.4 and the two versions of clang/clang++. The first one is the upstream LLVM/CLANG 19.0.0 while the other one is a customization of the version 17.0.0 to add more theadvector functionalities and better support this extension, done for the RuyiSDK by the same community. The Tab. 5.3 describe the compilation options/flags used with each of these compilers to run the test and the inference results showcased in Fig. 5.7 and Fig. 5.8.

In Tab. 5.3 it's clear that there is only one compiler that fully supports the

Compiler	Flags C/C++
Xuantie GCC 10.4	-mcpu=c920 -O3
GCC 13.2	-O3
Clang 19	-march=rv64gc_zfh_xtheadba_xtheadbb_xtheadbs _xtheadcmo_xtheadcondmov_xtheadfmemidx_xtheadmac _xtheadmemidx_xtheadmempair_xtheadsync -O3
Clang 17 Ruyisk	-march=rv64gc_zfh_xtheadba_xtheadbb_xtheadbs _xtheadcmo _xtheadcondmov_xtheadfmemidx_xtheadmac_xtheadmemidx _xtheadmempair_xtheadsync _xtheadvector_xtheadzvamo -O3

Table 5.3: Compilers used for testing and the different CFLAGS and CXXFLAGS during multithreading and compiler comparison

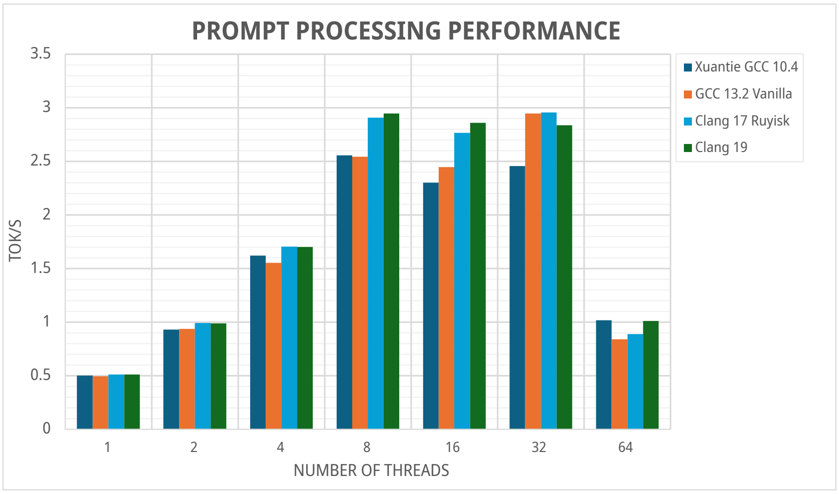


Figure 5.7: Prompt evaluation performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" multi-core, multi-threaded, varying the number of cores, compiled with Xuantie GCC 10.4, GCC 13.2 Vanilla, Clang 17 Ruyisk customization and Clang 19. Refer to Tab. 5.3 for the specific flags used.

specific CPU microarchitecture, the Xuantie GCC. For the rest we add all the extra extensions that are supported by the compiler and that our chip is compliant with, except for GCC 13.2 that is used as baseline with its default configuration for the machine. In this test, in accordance with other multiple tests and evaluations of as previously mentioned, clang showcases its superiority and even is capable of outperforming and make a better job than the vendor-customized GCC for token generation and most of cases.

Inside this competition, we can observe that, even if the customized version of Clang has some support for the vector instructions, leading to the possibility of

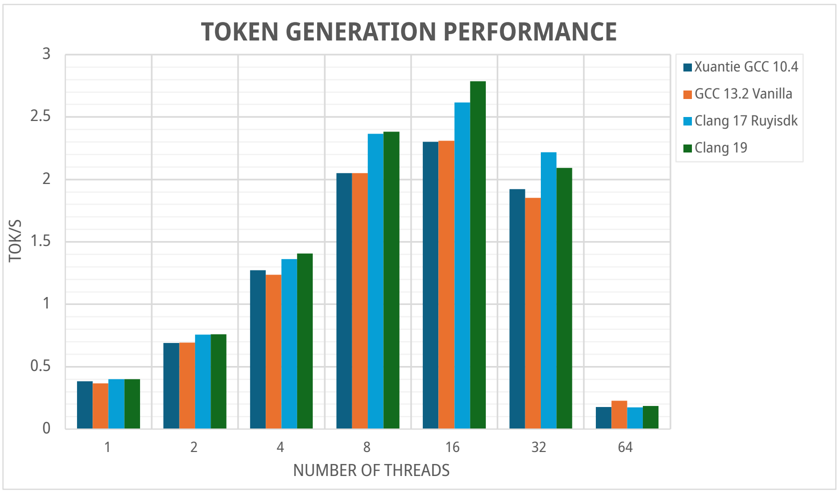


Figure 5.8: Token generation performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" multi-core, multi-threaded, varying the number of cores, compiled with Xuantie GCC 10.4, GCC 13.2 Vanilla, Clang 17 RuyiSDK customization and Clang 19. Refer to Tab. 5.3 for the specific flags used.

auto-vectorization optimizations, the upstream Clang is capable of outperform it in most of cases generating the best performance for token generation and prompt processing. With these results, we chose Clang 19 upstream version to further explore the capabilities of this hardware for inference as it was capable of generating the best results in both categories and mostly in token evaluation. The highest performance accomplished with this strategy is 2.79 tok/s for token generation and 2.96 tok/s for prompt processing.

The next exploration involved numa-awareness policies using the tool numactl in combination with the deactivation of the NUMA balancing policy of Linux. For this testing, we used two different strategies, one limiting the cores used, binding the execution to specific cores in the same NUMA region to reduce the latency in communication and data shared and the numactl interleave policy that uses a round-robing thread allocation strategy among the NUMA regions specified. The addition of the NUMA-aware execution completely changes the scalability of the system as shown in the Fig. 5.10, where the prompt processing continues to improve with the the increase in threads and in Fig. 5.9 where it also improves until 32 threads for token generation.

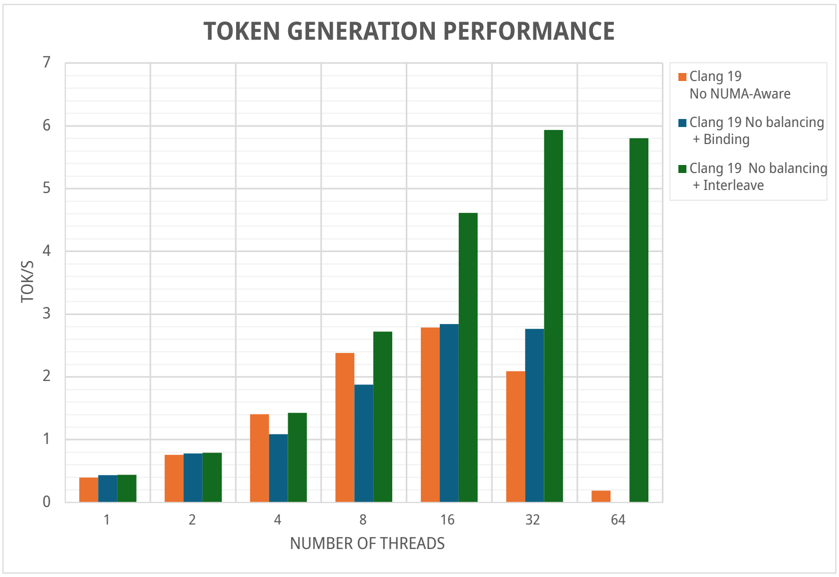


Figure 5.9: Token generation performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" multi-core, multi-threaded, varying the number of cores, compiled with Clang 19. Refer to Tab. 5.3 for the specific flags used.

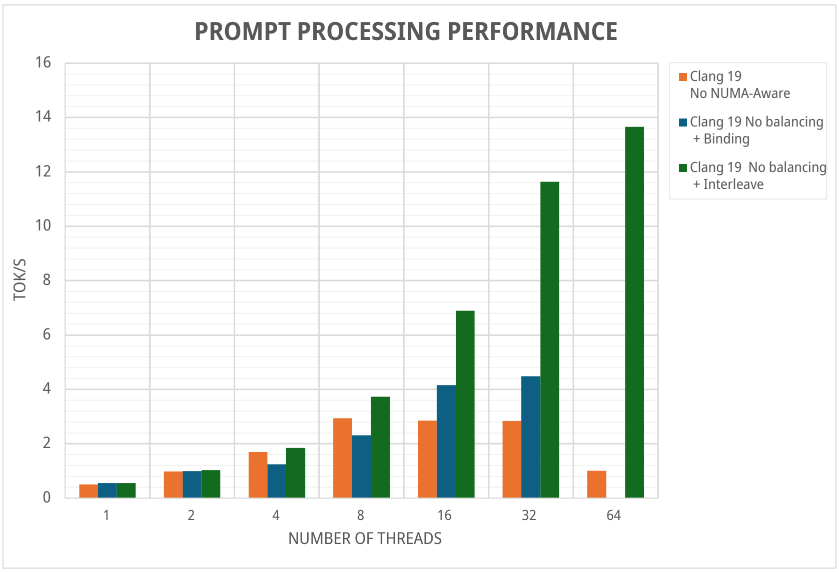


Figure 5.10: Prompt evaluation performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" multi-core, multi-threaded, varying the number of cores, compiled with Clang 19. Refer to Tab. 5.3 for the specific flags used.

Here, the best results are accomplished by disabling the NUMA balancing policy and allowing numactl to apply its interleave policy around all NUMA regions with a bigger number of threads, providing a maximum of 5.93 tok/s for token generation and 13.66 tok/sec for prompt processing. These results improve the previous Non-numa-aware best performances for token generation by 2.13 times and 4.61 times in the case of prompt processing.

In-depth analysis

Further exploration of the effect of NUMA awareness was done in order to obtain more in-depth knowledge about the effect of the policies and how they impact the execution in the machine. We used the case of 32 threads and analysed 5 different strategies: no numa-awareness (base), Disabling NUMA balancing policy, Disabling NUMA balancing and binding the cores to the NUMA regions 2 and 3, also binding to the same NUMA regions memory, and finally using the numaclt interleave across all regions. The results for token generations and prompt processing are shown in Fig. 5.11 where the performance is progressively improving with its modification in the strategy. During these executions, we also monitored with the Linux *perf* tool the inference metrics from the system. Tab. 5.4 reflects the corresponding values for the task time, context-switches, CPU migrations and page faults.

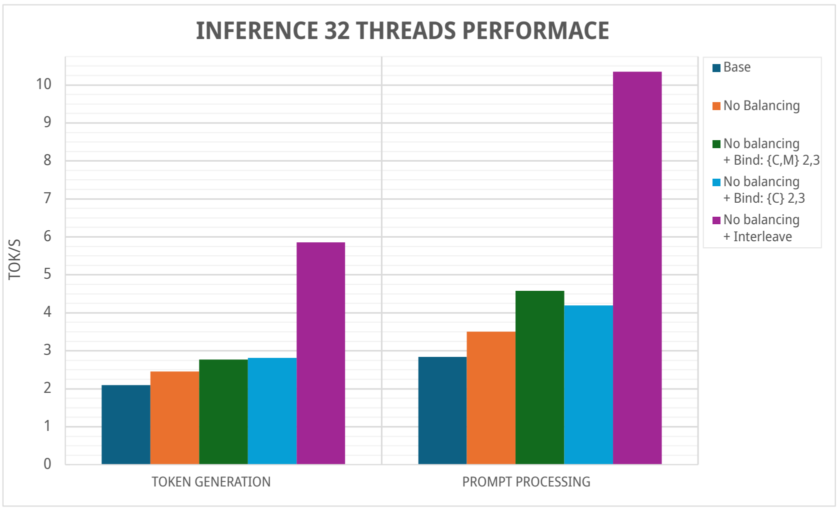


Figure 5.11: Inference performance with llama.cpp with model llama-2-7b.Q4_0 and prompt "Once upon a time, there was a kingdom" multi-core, 32 threads, varying stragegy for numa-awareness, NUMA balacing or none, compiled with Clang 19. Refer to Tab. 5.3 for the specific flags used.

The effectiveness of disabling the NUMA balancing is apparent as all the metrics, except execution time itself, get reduced by an order of magnitude. The core binding

Perf metrics	**task-clock (ms)**	**context-switches**	**cpu-migrations**	**page-faults**
Base	1,248,106	42,203	2,748	186,148
No Balancing	1,110,044	5,351	629	75,395
No balancing + Bind: {C,M} 2,3	978,231	4,704	9	77,083
No balancing + Bind: {C} 2,3	970,779	4,568	10	77,349
No balancing + Interleave all	441,827	3,172	84	75,529

Table 5.4: Perf basic metric of running

is clearly reflected in the number of CPU migrations, as they reduce dramatically. But besides these step-by-step improvements, the round-robin interleave policy reduces the context switches even more. This happens despite increasing the CPU migrations and not having a better number of page faults than just disabling the NUMA balancing policy. These results showcase the complexity in a system like the MILK-V Pioneer and the different metrics that play a role in various cases.

Chapter 6

Conclusion and Future Work

The current landscape of HPC is gaining more and more importance with the inclusion of complex computational task like the execution of LLMs and making these revolutionary technologies available for everyone. RISC-V is rapidly catching up as high-performance architecture, thriving thanks to its openness and customization possibilities that many initiatives worldwide try to exploit.

The inference process of LLMs is a great challenge nowadays due to its novelty and high demand of computational power and memory requirements. These problems added to the lack of maturity of RISC-V in more capable systems makes it even greater challenge.

During this thesis, we could observe the multi-dimensionality of these problems. Starting from the optimization in a heterogeneous computing system with vector capabilities such as the SG2042 at a single core level. Generating custom kernels that can target specific architectures and specific extensions such as the RISC-V Vector 0.7.1 specification extension lead to significant improvemts compared to scalar implementations, and even simpler vectorized implementations showed its superior performance. On the next upper hardware level, our work reflected the importance of the knowledge of the hardware organization, such as considering NUMA regions and their affinity and the Linux operating system optimizations. Current complex computing systems with hierarchical memories and sub-regions require a more in-depth awareness and combination with the actuall workflow of execution, and as the disabling of the NUMA balancing default Linux policy shown, the default configuration or execution can be far from the optimal setup for best performance. Finally, at a software ecosystem level it was displayed the difference among compilers so popular as GCC or LLVM/Clang and their customizations and the support of the RISC-V and RISC-V vector extensions, highlight the superiority in performance and optimization of LLVM/Clang. Other pieces of software that helped in the development were the numaclt, extensively used to gain more control over the thread allocation policy showing great utility of its built-in interleave more elaborated policy, and perf, key to obtain the metric that give significance to the changes in performance.

This multi-level approach allowed us to improve the performance of the machine progressively and analyse the effect of the different alternatives that could be used when targeting HPC tasks such as the inference with LLMs. These achievements were favoured by the availability of an open-source and efficient inference framework like llama.cpp. During this work, the different manners in which internal information can be obtained from modifying and analysing the inference framework played a crucial role in a better understanding on how the optimization should be targeted and carried.

All in all, we showcased the viability of inference with state-of-the-art models with a great performance and adapting to the current trends towards quantized models and only CPU execution. All of this is in a machine fully using a many-core CPU based only in RISC-V cores with a high level of complexity and relatively low support from a software ecosystem compared to classical computer architectures and HPC platforms.

This thesis could be extended by improving several areas of our testing. A customized thread allocation policy while being NUMA-aware to further improve the control over the multi-threading execution without the usage of the numactl tool. Other system-level optimization could include the usage of Hugepages to test the potential improvements of bigger pages as the hardware is compliant with it. Following, besides the hardware and operating system generic improvements, the inclusion of more efficient techniques during the inference related with the model, such as flash attention or more operation fusion could improve significantly the performance due to the important memory bottleneck that this hardware suffers from. One important improvement would be creating vectorized versions for the quantizing and dequantizing operations that are needed during the inference of quantized models. Finally, our implementation should also be adapted to the other quantization types.

Appendix A

Kernel Source Code

Listing A.1: function ggml_compute_forward_mul_mat() with added comments in ggml.c

```c


#include <riscv_vector.h>
#include <stdio.h>
#include <string.h>

#define QK8_0 32
typedef struct {
    _Float16   d;          // delta
    int8_t   qs[QK8_0];   // quants
} block_q8_0;

#define QK4_0 32
typedef struct {
    _Float16   d;          // delta
    uint8_t qs[QK4_0 / 2];  // nibbles / quants
} block_q4_0;

inline float ggml_compute_fp16_to_fp32_zfh( const _Float16 h) {
    _Float16 tmp;
    memcpy(&tmp, &h, sizeof(_Float16));
    return (float)tmp;
}


void kernel_complex(int ne01,int ne0, const block_q4_0 *x_0,const
    block_q4_0 *x_1,  const block_q8_0 * y, float * dst_temp, int k,
    int h, int n_blocks ){
   float sumf_0 = 0.0;
   float sumf_1 = 0.0;
```

```
    for (int i = 0; i < n_blocks; i+=2){
      const int vl = __riscv_vsetvl_e8m1(32/2); //QK4_0/2 = qk = 32
      //kernel_in(ne01, x_0_1, x_0_0, x_1_0,x_1_1 , y_0 ,y_1 , &sumf_0 , &sumf_1, x_00_d, x_01_d, x_10_d, x_11_d,  y_0_d,  y_1_d);
      vuint8m1_t tx_0_0 = __riscv_vle8_v_u8m1(x_0[i].qs, vl); //32 int4 values weights
      vuint8m1_t tx_0_1 = __riscv_vle8_v_u8m1(x_0[i+1].qs, vl);
      vuint8m1_t tx_1_0 = __riscv_vle8_v_u8m1(x_1[i].qs, vl); //32 int4 values weights
      vuint8m1_t tx_1_1 = __riscv_vle8_v_u8m1(x_1[i+1].qs, vl);
      vint8m1_t y0_0 = __riscv_vle8_v_i8m1(y[i].qs, vl); //16 int8 values
       vint8m1_t y0_1 = __riscv_vle8_v_i8m1(y[i].qs+16, vl); //16 int8 values
       vint8m1_t y1_0 = __riscv_vle8_v_i8m1(y[i+1].qs, vl); //16 int8 values
       vint8m1_t y1_1 = __riscv_vle8_v_i8m1(y[i+1].qs+16, vl); //16 16 int8 values

     //DEQUANTIZING
      //SPLITING THE INT8 INTO 2 INT4
        vuint8m1_t x_a_0_0 = __riscv_vand_vx_u8m1(tx_0_0, 0x0F, vl);
        vuint8m1_t x_l_0_0 = __riscv_vsrl_vx_u8m1(tx_0_0, 0x04, vl);
        vuint8m1_t x_a_0_1 = __riscv_vand_vx_u8m1(tx_0_1, 0x0F, vl);
        vuint8m1_t x_l_0_1 = __riscv_vsrl_vx_u8m1(tx_0_1, 0x04, vl);
        vuint8m1_t x_a_1_0 = __riscv_vand_vx_u8m1(tx_1_0, 0x0F, vl);
        vuint8m1_t x_l_1_0 = __riscv_vsrl_vx_u8m1(tx_1_0, 0x04, vl);
        vuint8m1_t x_a_1_1 = __riscv_vand_vx_u8m1(tx_1_1, 0x0F, vl);
        vuint8m1_t x_l_1_1 = __riscv_vsrl_vx_u8m1(tx_1_1, 0x04, vl);

         //Reinterpret
         vint8m1_t x_ai_0_0 = __riscv_vreinterpret_v_u8m1_i8m1(x_a_0_0);
         vint8m1_t x_li_0_0 = __riscv_vreinterpret_v_u8m1_i8m1(x_l_0_0);
         vint8m1_t x_ai_0_1 = __riscv_vreinterpret_v_u8m1_i8m1(x_a_0_1);
         vint8m1_t x_li_0_1 = __riscv_vreinterpret_v_u8m1_i8m1(x_l_0_1);
         vint8m1_t x_ai_1_0 = __riscv_vreinterpret_v_u8m1_i8m1(x_a_1_0);
         vint8m1_t x_li_1_0 = __riscv_vreinterpret_v_u8m1_i8m1(x_l_1_0);
         vint8m1_t x_ai_1_1 = __riscv_vreinterpret_v_u8m1_i8m1(x_a_1_1);
         vint8m1_t x_li_1_1 = __riscv_vreinterpret_v_u8m1_i8m1(x_l_1_1);

         vint8m1_t vxa_0_0 = __riscv_vadd_vx_i8m1(x_ai_0_0, -8, vl);
        vint8m1_t vxl_0_0 = __riscv_vadd_vx_i8m1(x_li_0_0, -8, vl);
        vint8m1_t vxa_0_1 = __riscv_vadd_vx_i8m1(x_ai_0_1, -8, vl);
        vint8m1_t vxl_0_1 = __riscv_vadd_vx_i8m1(x_li_0_1, -8, vl);
        vint8m1_t vxa_1_0 = __riscv_vadd_vx_i8m1(x_ai_1_0, -8, vl);
        vint8m1_t vxl_1_0 = __riscv_vadd_vx_i8m1(x_li_1_0, -8, vl);
        vint8m1_t vxa_1_1 = __riscv_vadd_vx_i8m1(x_ai_1_1, -8, vl);
        vint8m1_t vxl_1_1 = __riscv_vadd_vx_i8m1(x_li_1_1, -8, vl);

        //Now that the have the weights we do int8*int8 widening and
    then we accumunate
        vint16m2_t vec_mul_0_0 = __riscv_vwmul_vv_i16m2(vxa_0_0, y0_0
    , vl);
        vint16m2_t vec_macc_0_0 =__riscv_vwmacc_vv_i16m2(vec_mul_0_0,
    vxl_0_0, y0_1, vl);
        vint16m2_t vec_mul_0_1 = __riscv_vwmul_vv_i16m2(vxa_0_1, y1_0
    , vl);
        vint16m2_t vec_macc_0_1 =__riscv_vwmacc_vv_i16m2(vec_mul_0_1,
    vxl_0_1, y1_1, vl);

        vint16m2_t vec_mul_1_0 = __riscv_vwmul_vv_i16m2(vxa_1_0, y0_0
    , vl);
        vint16m2_t vec_macc_1_0 =__riscv_vwmacc_vv_i16m2(vec_mul_1_0,
    vxl_1_0, y0_1, vl);
        vint16m2_t vec_mul_1_1 = __riscv_vwmul_vv_i16m2(vxa_1_1, y1_0
    , vl);
        vint16m2_t vec_macc_1_1 = __riscv_vwmacc_vv_i16m2(vec_mul_1_1
    ,vxl_1_1, y1_1, vl);

        const vint32m1_t vec_zero = __riscv_vmv_v_x_i32m1(0,16);

        //In RVV 0.7.1 IT IS REQUIRED A MASK
        //We add all the results (the results are spread in 1 vector
    register) of the same Quantized blocks in 1 number
        vint32m1_t v_res_0_0 = __riscv_vwredsum_vs_i16m2_i32m1(
    vec_macc_0_0, vec_zero, vl);
        vint32m1_t v_res_0_1 = __riscv_vwredsum_vs_i16m2_i32m1(
    vec_macc_0_1, vec_zero, vl);

        vint32m1_t v_res_1_0 = __riscv_vwredsum_vs_i16m2_i32m1(
    vec_macc_1_0, vec_zero, vl);
        vint32m1_t v_res_1_1 = __riscv_vwredsum_vs_i16m2_i32m1(
    vec_macc_1_1, vec_zero, vl);

        int sumi_0_0 = __riscv_vmv_x_s_i32m1_i32(v_res_0_0);
        int sumi_0_1 = __riscv_vmv_x_s_i32m1_i32(v_res_0_1);
        int sumi_1_0 = __riscv_vmv_x_s_i32m1_i32(v_res_1_0);
        int sumi_1_1 = __riscv_vmv_x_s_i32m1_i32(v_res_1_1);

       // Y dequantizing weight upscaling
```

```
            float y_d_0 = ggml_compute_fp16_to_fp32_zfh(y[i].d);
            float y_d_1 = ggml_compute_fp16_to_fp32_zfh(y[i+1].d);

            //We multiply each result, by its dequantizing scale of x (4 blocks ) and y (2blocks)
            sumf_0 += sumi_0_0*ggml_compute_fp16_to_fp32_zfh(x_0[i].d)*y_d_0 + sumi_0_1*ggml_compute_fp16_to_fp32_zfh(x_0[i+1].d)*y_d_1;
            sumf_1 += sumi_1_0*ggml_compute_fp16_to_fp32_zfh(x_1[i].d)*y_d_0 + sumi_1_1*ggml_compute_fp16_to_fp32_zfh(x_1[i+1].d)*y_d_1;
            }    //nb = number of bytes per row
             //Finally data storage
            *(dst_temp+h+k*ne0) = sumf_0;
            *(dst_temp+(h+1)+k*ne0) = sumf_1;
    }
```

Bibliography

[1] Zhibo Chu, Shiwen Ni, Zichong Wang, Xi Feng, Min Yang, and Wenbin Zhang. *History, Development, and Principles of Large Language Models-An Introductory Survey.* arXiv:2402.06853 [cs]. 2024. URL: `http://arxiv.org/abs/2402.06853` (visited on 08/14/2024) (cit. on pp. 3, 4, 8).

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space.* 2013. arXiv: `1301.3781 [cs.CL]`. URL: `https://arxiv.org/abs/1301.3781` (cit. on p. 3).

[3] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview.* 2019. arXiv: `1912.05911 [cs.LG]`. URL: `https://arxiv.org/abs/1912.05911` (cit. on pp. 4, 6).

[4] Gang Chen. *A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation.* 2018. arXiv: `1610.02583 [cs.LG]`. URL: `https://arxiv.org/abs/1610.02583` (cit. on p. 4).

[5] Wikimedia Commons. *Recurrent Neural Network Unfold.* Accessed: 2024-08-24. 2021. URL: `https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg` (cit. on p. 5).

[6] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Comput.* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735` (cit. on p. 4).

[7] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning.* `https://D2L.ai`. Cambridge University Press, 2023 (cit. on pp. 4, 6).

[8] M. Schuster and K.K. Paliwal. «Bidirectional recurrent neural networks». In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681. DOI: `10.1109/78.650093` (cit. on p. 4).

[9] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks.* 2014. arXiv: `1409.3215 [cs.CL]`. URL: `https://arxiv.org/abs/1409.3215` (cit. on p. 5).

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate.* 2016. arXiv: `1409.0473 [cs.CL]`. URL: `https://arxiv.org/abs/1409.0473` (cit. on p. 7).

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need.* 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762` (cit. on pp. 6, 7, 9).

[12] Xiao Fu, Weiling Yang, Dezun Dong, and Xing Su. «Optimizing Attention by Exploiting Data Reuse on ARM Multi-core CPUs». In: *Proceedings of the 38th ACM International Conference on Supercomputing.* ICS '24. Kyoto, Japan: Association for Computing Machinery, 2024, pp. 137–149. ISBN: 9798400706103. DOI: `10.1145/3650200.3656620`. URL: `https://doi.org/10.1145/3650200.3656620` (cit. on p. 10).

[13] Tong Xiao and Jingbo Zhu. *Introduction to Transformers: an NLP Perspective.* 2023. arXiv: `2311.17633 [cs.CL]`. URL: `https://arxiv.org/abs/2311.17633` (cit. on p. 10).

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* 2019. arXiv: `1810.04805 [cs.CL]`. URL: `https://arxiv.org/abs/1810.04805` (cit. on p. 8).

[15] Alec Radford and Karthik Narasimhan. *Improving Language Understanding by Generative Pre-Training.* 2018. URL: `https://api.semanticscholar.org/CorpusID:49313245` (cit. on p. 8).

[16] Enfang Cui, Tianzheng Li, and Qian Wei. «RISC-V Instruction Set Architecture Extensions: A Survey». In: *IEEE Access* 11 (2023), pp. 24696–24711. DOI: `10.1109/ACCESS.2023.3246491` (cit. on pp. 10–12).

[17] RISC-V International Foundation. *RISC-V Specification Lifecycle Guide.* Accessed: 2024-09-01. 2024. URL: `https://riscv.org/specifications` (cit. on p. 12).

[18] RISC-V International. *The RISC-V Instruction Set Manual Repository.* 2024. URL: `https://github.com/riscv/riscv-isa-manual` (cit. on p. 13).

[19] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools.* pearson Education, 2007 (cit. on pp. 15, 16).

[20] Fred J. Pollack. «New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only)». In: *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture.* MICRO 32. Haifa, Israel: IEEE Computer Society, 1999, p. 2. ISBN: 076950437X (cit. on p. 17).

[21] Daniel Etiemble. «45-year CPU evolution: one law and two equations». In: *arXiv preprint arXiv:1803.00254* (2018) (cit. on p. 17).

[22] Karl Rupp. *Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. Accessed: 2024-09-19. 2024 (cit. on p. 17).

[23] Intel Corporation David Ott. *Optimizing Applications for NUMA*. Accessed: 2024-06-05. 2011. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf (cit. on p. 18).

[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. «Basic Linear Algebra Subprograms for Fortran Usage». In: *ACM Trans. Math. Softw.* 5.3 (1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: https://doi.org/10.1145/355841.355847 (cit. on pp. 19, 20).

[25] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. «An extended set of FORTRAN basic linear algebra subprograms». In: *ACM Trans. Math. Softw.* 14.1 (1988), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/42288.42291. URL: https://doi.org/10.1145/42288.42291 (cit. on p. 21).

[26] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. «A set of level 3 basic linear algebra subprograms». In: *ACM Trans. Math. Softw.* 16.1 (1990), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: https://doi.org/10.1145/77626.79170 (cit. on p. 21).

[27] R.C. Whaley and J.J. Dongarra. «Automatically Tuned Linear Algebra Software». In: *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 1998, pp. 38–38. DOI: 10.1109/SC.1998.10004 (cit. on p. 22).

[28] Kazushige Goto and Robert A. van de Geijn. «Anatomy of high-performance matrix multiplication». In: *ACM Transactions on Mathematical Software* 34.3 (2008), pp. 12–. DOI: 10.1145/1356052.1356053 (cit. on p. 22).

[29] Jianyu Huang and Robert A. van de Geijn. *BLISlab: A Sandbox for Optimizing GEMM*. 2016. arXiv: 1609.00076 [cs.MS]. URL: https://arxiv.org/abs/1609.00076 (cit. on p. 23).

[30] Florian Zaruba, Fabian Schuiki, and Luca Benini. *Manticore: A 4096-core RISC-V Chiplet Architecture for Ultra-efficient Floating-point Computing*. 2020. arXiv: 2008.06502 [cs.AR]. URL: https://arxiv.org/abs/2008.06502 (cit. on pp. 24–26).

[31] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. «Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads». In: *IEEE Transactions on Computers* 70.11 (2021), pp. 1845–1860. ISSN: 2326-3814. DOI: `10.1109/tc.2020.3027900`. URL: `http://dx.doi.org/10.1109/TC.2020.3027900` (cit. on p. 24).

[32] F. Zaruba and L. Benini. «The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. ISSN: 1557-9999. DOI: `10.1109/TVLSI.2019.2926114` (cit. on p. 24).

[33] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. «Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores». In: *IEEE Transactions on Computers* 70.2 (2021), pp. 212–227. DOI: `10.1109/TC.2020.2987314` (cit. on p. 25).

[34] Chen Chen et al. «Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product». In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 52–64. DOI: `10.1109/ISCA45697.2020.00016` (cit. on pp. 26–28).

[35] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. *BOOM v2: an open-source out-of-order RISC-V core*. Tech. rep. UCB/EECS-2017-157. EECS Department, University of California, Berkeley, 2017. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html` (cit. on p. 26).

[36] Andreas Traber, Michael Gautschi, and Pasquale Davide Schiavone. *RI5CY: User Manual*. Licensed under the Solderpad Hardware License, Version 0.51. Micrel Lab, Multitherman Lab, University of Bologna, Italy, and Integrated Systems Lab, ETH Zürich, Switzerland. 2019. URL: `http://solderpad.org/licenses/SHL-0.51` (cit. on p. 26).

[37] Milk-V. *Pioneer Overview*. Accessed: 2024-06-20. 2024. URL: `https://milkv.io/docs/pioneer/overview` (cit. on p. 29).

[38] Chao Wei. *CPU Sophon SG2042 Technical Reference Manual*. Technical Reference Manual. Sophon. 2024 (cit. on p. 30).

[39] Georgi Gerganov. *llama.cpp: LLM inference in C/C++*. `https://github.com/ggerganov/llama.cpp`. Accessed: 2024-09-12. 2023 (cit. on pp. 31, 33).

[40] GNU Community. *GNU C Compiler Internals*. Wikibooks, 2024. URL: https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals (visited on 06/11/2024) (cit. on p. 35).

[41] Yafan Huang. *Getting Started with LLVM*. https://hyfshishen.github.io/tutorial-01-llvm.html. Accessed: 2024-08-01. 2024 (cit. on p. 36).

[42] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971 (cit. on p. 36).

[43] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288 (cit. on pp. 37, 38).

[44] Zhang Xianyi, Wang Qian, and Zhang Yunquan. «Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor». In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 2012, pp. 684–691. DOI: 10.1109/ICPADS.2012.97 (cit. on p. 38).

[45] OpenBLAS Team. *OpenBLAS FAQ*. Retrieved 2024-10-11. 2024. URL: https://github.com/OpenMathLib/OpenBLAS/wiki/Faq (cit. on p. 39).

[46] Christian Fibich, Stefan Tauner, Peter Rössler, and Martin Horauer. «Evaluation of Open-Source Linear Algebra Libraries targeting ARM and RISC-V Architectures». In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. 2020, pp. 663–672. DOI: 10.15439/2020F145 (cit. on p. 39).

[47] *RuyiSDK: Integrated Development Environment for RISC-V*. https://ruyisdk.org/. Accessed: 2024-10-11 (cit. on p. 42).

[48] Nick Brown, Maurice Jamieson, Joseph Lee, and Paul Wang. «Is RISC-V ready for HPC prime-time: Evaluating the 64-core Sophon SG2042 RISC-V CPU». In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Vol. 2021. SC-W 2023. ACM, Nov. 2023, pp. 1566–1574. DOI: 10.1145/3624062.3624234. URL: http://dx.doi.org/10.1145/3624062.3624234 (cit. on p. 42).

[49] Gurbinder Gill and Ramesh V. Peri. «Measure the Impact of NUMA Migrations on Performance». In: *Intel Corporation* (2019). Accessed: 2024-09-14. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/measure-the-impact-of-numa-migrations-on-performance.html (cit. on p. 45).

[50] Xiao Fu, Weiling Yang, Dezun Dong, and Xing Su. «Optimizing Attention by Exploiting Data Reuse on ARM Multi-core CPUs». In: *Proceedings of the 38th ACM International Conference on Supercomputing*. 2024, pp. 137–149 (cit. on p. 47).

[51] David Bindel. *Matrix Multiplication Optimization*. `https://github.com/cs5220-f20/matmul-project`. Accessed: 2024-06-15. 2024 (cit. on p. 48).

[52] Francisco Igual, Luis Piñuel, Sandra Catalán, Héctor Martínez, Adrián Castelló, and Enrique Quintana-Ortí. «Automatic Generation of Micro-kernels for Performance Portability of Matrix Multiplication on RISC-V Vector Processors». In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 1523–1532. ISBN: 9798400707858. DOI: `10.1145/3624062.3624229`. URL: `https://doi.org/10.1145/3624062.3624229` (cit. on p. 49).

[53] Github) Anonymous(camel-cdr. *The Milk-V Pioneer: Performance Analysis and Benchmarking*. `https://example.com/milk-v-pioneer`. Accessed: 2024-05-02. 2024 (cit. on p. 49).

[54] Jieyang Chen et al. «TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs». In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 106–116. ISBN: 9781450360791. DOI: `10.1145/3330345.3330355`. URL: `https://doi.org/10.1145/3330345.3330355` (cit. on p. 51).