Exploring High-Performance and Energy-Efficient Architectures for Edge AI-Enabled Applications

Présentée le 2 février 2024
Faculté des sciences et techniques de l'ingénieur
Laboratoire des systèmes embarqués
Programme doctoral en génie électrique
pour l'obtention du grade de Docteur ès Sciences
par
Joshua Alexander Harrison KLEIN

Acceptée sur proposition du jury
Prof. G. De Micheli, président du jury
Prof. D. Atienza Alonso, Dr M. Zapater Sancho, directeurs de thèse
Prof. K. Olcoz, rapporteuse
Dr S. Nassif, rapporteur
Prof. A. Burg, rapporteur
"If you torture the data long enough, it will confess to anything."
— Ronald Coase

To my family,
de facto and otherwise.
Acknowledgements

They say it takes a village to raise a child. I contend that it also takes a village to raise a doctoral student, as without the friendship, guidance, and support of so many others around me, I would not have been able to write this thesis.

I would like to first and foremost acknowledge and thank my thesis directors, Professor David Atienza and Professor Marina Zapater. The combination of David’s cultivation of ESL and long-term research strategizing, as well as Marina’s attentiveness towards the day-to-day and week-to-week research goals, always ensured that I was making regular and steady progress in my work. In particular I am especially grateful to both for always unconditionally encouraging and supporting me, regardless of my personal circumstances, paper rejections, and all of the difficulties faced over my studies. It was this unconditional will to see me succeed that first convinced me to join ESL as a doctoral student. As of writing this, four years later, I can confidently say I have no regrets in making that decision.

Next, I thank my thesis jury members, Giovanni De Micheli, Andreas Burg, Katzalin Olcoz, and Sani Nassif, for their collective expertise and critical eye to help ensure my thesis is the best it can be, and that its research contributions extend throughout the field as a whole.

Of course, I would be remiss if I did not thank my supervising research scientist, Giovanni Ansaloni, for not only helping to review very early drafts of the thesis, but for lending his support throughout my studies. It was his keen eye for research opportunities and new directions that helped to strategize and narrow my research, while simultaneously bridging my expertise with that of hardware designers and technologists.

I would also like to thank my initial doctoral mentor in ESL, Yasir Qureshi, who trained me in using gem5-X and whose calm and collected demeanor helped me tackle some of the most difficult technical challenges I faced during the early years of my work.

I share this thanks with my direct collaborators, Alexandre Levisse, Rafael Medina Morillas, Alireza Amirshahi, Darong Huang, Irem Boybat, Abu Sebastian, David Truan, Gregory Medwed, Gabriel Catel Torres, Karan Pathak, and Flavio Ponzina, whose expertise, camaraderie, and shared desire to pursue knowledge together led to the works that make up this thesis. It is only through our partnerships and collaborations that this work had the background and foundation for which it could propel the field.

These collaborations would be impossible without ESL’s support staff, and so I would also
like to thank John Maxwell, Mikael Doche, Christoph Müller, and Homeira Salimi for their administrative and technical support throughout my doctoral studies.

Of course, all of the aforementioned are a small part of what I would call my "EPFL family", and so I further extend my thanks to Lara Orlandić, Denisa-Andreea Constantinescu, Rubén Rodriguez Álvarez, Stefano Albini, Christodoulos Kechris, Juan Saprizá, Grégoire Surrel, Iván Nyhanitra, Elisabetta de Giovanni, Artem Andreev, Dimitrios Samakovlis, Maven Zanoli, Clément Choné, Grégoire Eggermann, Amirhossein Shahbazinia, Renato Zanetti, Pengbo Yu, Una Pale, as well as Nikolina Tomic, for all of the hiking, cocktail parties, dog walking, adventuring, and dinners we shared.

My ESL family is but one of many families that contributed to my ability to go so far in life. Next, I would like to thank my Swiss family, Zinette Erkmann, as well as her direct family Nicolas, Harkon, Alexandra, Adam, Éthan, Grégoire, Django, Gültön, Jesse, and Mehmet-Ali, for adopting and welcoming me into their culture. It is you who made me feel truly at home in Switzerland. I also thank Julianna Dragos, whose guiding hand helped me grow emotionally and intellectually as I struggled with my adjustment to both becoming an adult and a doctoral student. It is because of our sessions that I have grown to appreciate who I am and what I have become, as well as recognize ways of improving myself over the long term.

I thank my Boston family, Nevin Zheng, Amy Hu, Yongnian Zheng, Lijun Yang, as well as my friends Eddy Luo and Sami Shahin, for seeing me through my Bachelor's and Master's degree, as well as cheering me on throughout my doctoral studies. For putting me on this path towards an engineering doctorate to begin with, I also thank my early influences in Engineering from Boston University, Ayse Coskun, James "Greg" McDaniel, and Aditya Narayan. It is you who not only showed me that I can do a doctoral degree, but that it would be fun and enriching as well.

I thank my best friend Ashley Kaufmann, and her associates Cooper Kaufmann, Ginger Kaufmann, Flüf Kaufmann, and Jasper Kaufmann, for being there for me through so many stages of my life, helping me to adjust my transition into adulthood, then university studies, then doctoral studies, and even through to my first job.

Lastly, I would like to thank my California family for their support throughout my studies. I thank my mother, Susanne Klein, for always encouraging and supporting my education no matter my status or stage in life, for always making my education and well-being your priority, even when I decided to move over 5000 miles away. Our weekly calls would regularly remind me of what I was working towards and that all you wanted for me was to live my life to its fullest.

I thank the elder of my sisters, Katherine Booth, for her love and support, as well as being the original inspiration to pursue higher education abroad. Growing up, I had always thought it would be impossible for me to achieve a similar level of education, achieve recognition in my field, and lead a more global life. And yet, it was always your example that reminded me I could succeed throughout my studies, even in the hardest of times.

I thank the younger of my sisters, Elizabeth Bryson, whose pragmatic and realistic perspective always kept me grounded in reality, set my expectations, and helped me realize my value as a person. It was your example that always encouraged me to challenge myself and pursue more
in life.
I thank the rest of my family, Matthew Klein, Neil Booth, Martyn Bryson, Imogen Booth, Amelia Booth, Zero Bryson, Maddie Booth, Eddy Booth, Lily Booth, Luna Booth, Jasmin Resurrection, Chris Resurrection, Olive Resurrection, Fiona Resurrection, and Iroh Resurrection for their love and support as well.
And last but not least, I would like to thank Marceline and Lohse Klein, who kept me mentally, emotionally, and physically in shape throughout my studies.
It is because of all these people and more that no matter where I am, I will always have love and support around me.

_Lausanne, Novembre 2023_
Abstract

The desire and ability to place AI-enabled applications on the edge has grown significantly in recent years. However, the compute-, area-, and power-constrained nature of edge devices are stressed by the needs of the AI-enabled applications, due to a general pressure to increase the size, depth, and capability of the underlying neural networks. These applications represent a worst-case scenario for numerous architecture-based problems due to their tendency to have a large memory footprint with millions or billions of parameters and high compute requirements via matrix operations. To address architectural issues that arise, e.g., the memory wall, computer architects and engineers have developed numerous solutions, frameworks, and techniques for modeling architectural solutions for AI-enabled applications. Apparatuses for simulating a variety of specialized systems with in-memory compute architectures, SIMD co-processors, neural network engines, and more, have all been proposed and implemented to varying degrees. However, many of these apparatuses suffer from a common limitation: they are designed with a very constrained set of experiments in mind, often only comparing their solutions against “conventional” systems. As a result, given a wide range of architectural choices for AI-enabled applications, it is extremely difficult to look at solutions both in isolation with respect to one another, as well as compare heterogeneous solutions that may employ one or several solutions simultaneously. The lack of traversability in architecture design-space explorations is a hindrance to future architecture development due to the complexity of architectural challenges to modern computing and the gaining popularity of heterogeneity in modern compute systems.

Therefore, in this doctoral thesis, I present the ALPINE framework: a full system-level computer architecture framework built atop the gem5-X simulator, and my accumulated work to build apparatuses, tools, and methodologies for implementing, modeling, and extracting vital statistics from new heterogeneous edge architectures for modern AI-enabled applications. By building a framework for modeling numerous novel accelerators and interfaces into a kernel-capable full system-level computer architecture simulation of general purpose systems, I am able to perform numerous architectural explorations for modern neural networks, as well as set up the basis for future explorations.

With ALPINE, I am able to first model real analog in-memory compute tiles for inference close to the CPU and interfaced via ISA extension. These tiles are able to perform matrix-vector
multiplication operations, a common bottleneck in AI-enabled applications, in constant
time. I conduct a wide variety of explorations of AI-enabled applications, including of Multi-
Layer Perceptrons, Recurrent Neural Networks, and Convolutional Neural Networks. By
looking at sub-regions-of-interest, we are able to extrapolate the benefits of using this solution
for AI-enabled applications, show cross-core communication as an emerging bottleneck,
and ultimately show up to 20.5x speedup and 20.8x energy gains over conventional SIMD-
vectorized systems. To address the communications bottleneck, we are then able to model,
within ALPINE, wireless scratchpad memories that can send data between cores at a rapid rate,
leading to an additional 20% performance improvement in the tested convolutional neural
networks.
Owing to the highly extensible nature of the ALPINE framework, the accelerator model was
also refactored to model small systolic arrays close to the CPU. Using Transformers as an
initial test case, we are able to show up to 89.5x application-wide speed-ups across Trans-
former models. With both a systolic array model and analog in-memory compute model
implemented in ALPINE, I am able to compare directly competing AI solutions in common
AI-enabled applications by weighing the trade-offs of capabilities, speedups, and energy gains
of two accelerator-based solutions in the same system, and also against systems with a SIMD
vectorized co-processor and larger caches.
To add another dimension of design-space heterogeneity to ALPINE, I perform additional
studies of the analog in-memory compute tiles that are placed far away from the CPU with a
bus-based data transfer interface and compare this directly with near-CPU cases and interfaces.
This exploration reveals the role of load-balancing and data transfer overheads in multi-
processed neural networks, and thus by tuning state-of-the-art load balancing algorithms,
we can achieve additional speedups and energy gains (up to over 5x) in a wide variety of
The studies presented in this thesis reveal numerous techniques and methodologies for not
only conducting heterogeneous architectural explorations, but also showing the practical
and pragmatic effects of implementing applications onto accelerators, isolating bottlenecks,
leveraging architectural tools and statistics, and more. ALPINE is ultimately generic enough
that future accelerator models and configurations, as well as future neural networks and
bottlenecks, can be efficiently explored.

Key words: full system-level simulation, computer architecture, AI, machine learning, deep
learning, heterogeneous architectures, neural networks, inference, in-memory computing, ac-
celerators, multi-layer perceptrons, recurrent neural networks, convolutional neural networks,
transformers, interfaces.
Résumé

Le désir et la capacité de placer des applications basées sur l’IA à la périphérie se sont considérablement accrues ces dernières années. Toutefois, les contraintes de calcul, de surface et d’énergie des dispositifs périphériques sont mises à l’épreuve par les besoins des applications basées sur l’IA, en raison d’une pression générale visant à augmenter la taille, la profondeur et la capacité des réseaux neuronaux sous-jacents. Ces applications représentent le pire scénario pour de nombreux problèmes d’architecture, car elles ont tendance à avoir une grande empreinte mémoire avec des millions ou des milliards de paramètres et des exigences de calcul élevées par le biais d’opérations matricielles. Pour résoudre les problèmes architecturaux qui se posent, par exemple le mur de mémoire, les architectes et ingénieurs informatiques ont mis au point de nombreuses solutions, cadres et techniques pour modéliser des solutions architecturales pour les applications basées sur l’IA. Des appareils permettant de simuler une variété de systèmes spécialisés avec des architectures de calcul en mémoire, des coprocesseurs SIMD, des moteurs de réseaux neuronaux, etc. ont tous été proposés et mis en œuvre à des degrés divers. Toutefois, nombre de ces appareils souffrent d’une limitation commune : ils sont conçus avec un ensemble très restreint d’expériences à l’esprit, comparant souvent leurs solutions uniquement à des systèmes "conventionnels". Par conséquent, compte tenu du large éventail de choix architecturaux pour les applications basées sur l’IA, il est extrêmement difficile d’examiner les solutions isolément les unes par rapport aux autres, ainsi que de comparer des solutions hétérogènes qui peuvent utiliser une ou plusieurs solutions simultanément. Le manque d’accessibilité dans les explorations de l’espace de conception de l’architecture est un obstacle au développement futur de l’architecture en raison de la complexité des défis architecturaux de l’informatique moderne et de la popularité croissante de l’hétérogénéité dans les systèmes de calcul modernes.

Par conséquent, dans cette thèse de doctorat, je présente le cadre ALPINE : un cadre d’architecture informatique au niveau du système complet construit sur le simulateur gem5-X, et mon travail accumulé pour construire des appareils, des outils et des méthodologies pour mettre en œuvre, modéliser et extraire des statistiques vitales des nouvelles architectures hétérogènes de pointe pour les applications modernes basées sur l’IA. En construisant un cadre pour la modélisation de nombreux accélérateurs et interfaces novateurs dans une simulation d’architecture informatique complète au niveau du noyau de systèmes à usage général, je suis en mesure d’effectuer de nombreuses explorations architecturales pour les réseaux neuronaux.
modernes, ainsi que d’établir les bases de futures explorations.

Avec ALPINE, je peux d’abord modéliser de véritables tuiles de calcul analogiques en mémoire pour l’inférence, proches de l’unité centrale et interfacées via une extension ISA. Ces tuiles sont capables d’effectuer des opérations de multiplication matrice-vecteur, un goulot d’étranglement commun dans les applications basées sur l’IA, en temps constant. Je mène une grande variété d’explorations d’applications basées sur l’IA, y compris des perceptrons multicouches, des réseaux neuronaux récurrents et des réseaux neuronaux convolutifs. En examinant les sous-régions d’intérêt, nous sommes en mesure d’extrapoler les avantages de l’utilisation de cette solution pour les applications basées sur l’IA, de montrer que la communication entre les cœurs est un goulot d’étranglement émergent et, en fin de compte, de montrer jusqu’à 20,5 fois la vitesse et 20,8 fois les gains énergétiques par rapport aux systèmes SIMD-vectorisés conventionnels. Pour résoudre le goulot d’étranglement des communications, nous sommes alors en mesure de modéliser, dans ALPINE, des mémoires scratchpad sans fil qui peuvent envoyer des données entre les cœurs à un rythme rapide, ce qui entraîne une amélioration supplémentaire des performances de 20% dans les réseaux neuronaux convolutifs testés.

Grâce à la nature hautement extensible du cadre ALPINE, le modèle d’accélérateur a également été remanié pour modéliser de petits réseaux systoliques proches de l’unité centrale. En utilisant les transformateurs comme cas de test initial, nous sommes en mesure de montrer des accélérations allant jusqu’à 89,5 fois pour l’ensemble de l’application à travers les modèles de transformateurs. Avec un modèle de réseau systolique et un modèle de calcul analogique en mémoire implémentés dans ALPINE, je suis en mesure de comparer des solutions d’IA directement concurrentes dans des applications courantes basées sur l’IA en pesant les compromis de capacités, de gains de vitesse et d’énergie de deux solutions basées sur des accélérateurs dans le même système, et également par rapport à des systèmes avec un coprocesseur vectorisé SIMD et des caches plus importants.

Pour ajouter une autre dimension d’hétérogénéité de l’espace de conception à ALPINE, j’effectue des études supplémentaires sur les tuiles de calcul analogiques en mémoire qui sont placées loin de l’unité centrale avec une interface de transfert de données basée sur un bus et je les compare directement avec des cas et des interfaces proches de l’unité centrale. Cette exploration révèle le rôle de l’équilibrage de la charge et des frais généraux de transfert de données dans les réseaux neuronaux multiprocessus. Ainsi, en ajustant les algorithmes d’équilibrage de la charge les plus modernes, nous pouvons obtenir des accélérations supplémentaires et des gains d’énergie (jusqu’à plus de 5 fois) dans une grande variété de réseaux neuronaux convolutifs.

Les études présentées dans cette thèse révèlent de nombreuses techniques et méthodologies permettant non seulement de mener des explorations architecturales hétérogènes, mais aussi de montrer les effets pratiques et pragmatiques de la mise en œuvre d’applications sur des accélérateurs, d’isoler les goulets d’étranglement, d’exploiter les outils architecturaux et les statistiques, et bien plus encore. En fin de compte, ALPINE est suffisamment générique pour que les futurs modèles et configurations d’accélérateurs, ainsi que les futurs réseaux neuronaux et goulets d’étranglement, puissent être explorés efficacement.
# Contents

Acknowledgements

Abstract (English/Français)

List of figures

List of tables

List of acronyms

1 Introduction

1.1 AI and Edge Systems .......................... 1
1.2 The Power and Memory Walls .................... 2
1.3 Exploring Heterogeneous AI Solutions ............ 3
1.4 Thesis Contributions ................................ 4
   1.4.1 Apparatuses and Tools: gXR5 and ALPINE Frameworks ............... 6
   1.4.2 Tightly-Coupled AI Accelerator Solutions .................. 7
   1.4.3 Loosely-Coupled AI Accelerator Solutions .................. 9

2 Apparatuses and Tools: gXR5 and ALPINE

2.1 Introduction .................................. 11
2.2 Background: Full system-level simulation, gem5-X, and Related Work .... 12
   2.2.1 Simulation Types and Trade-Offs ......................... 12
   2.2.2 Related Works: Simulators for Heterogeneous Architectural Explorations 14
   2.2.3 Predecessor gem5-X ................................ 15
2.3 gXR5: gem5 Extension for Linux-Capable Full System RISC-V Simulation .... 16
   2.3.1 gXR5 Full System Model and the SimpleBoard Platform ............ 17
   2.3.2 RISC-V Unprivileged Spec Instructions, Routines, and Virtual Memory 19
   2.3.3 gXR5 Interruptor and SoC Models .......................... 22
   2.3.4 gXR5 Validation .................................. 25
2.4 The ALPINE Framework: Expanding gem5-X and RISC models for exploring Accelerators and Interfaces .................. 32
   2.4.1 Generic Peripheral Input-Output Device Implementation .......... 33
Chapter 0

2.4.2 Software Support: AIMClib and tinytensorlib .................... 34
2.4.3 Power Modeling .................................................. 35
2.5 Summary ............................................................. 35

3 Tightly-Coupled AI Accelerator Solutions 37
3.1 Introduction .......................................................... 37
3.2 Related Work ........................................................ 39
  3.2.1 Analog In-Memory Compute Tiles .................................. 39
  3.2.2 Systolic Arrays for Edge AI Inference ............................... 40
3.3 Analog In-Memory Compute Tiles for Machine Learning Inference Workloads 40
  3.3.1 Analog In-Memory Compute Tiles .................................. 40
  3.3.2 AIMC Tile-Enabled System Architecture ............................. 42
  3.3.3 AIMC Tile-Enabled System Simulation with ALPINE ............... 46
  3.3.4 Tightly-Coupled AIMC Tile Exploration Setup and Methodology ...... 48
  3.3.5 Multi-Layer Perceptron Mappings and Analyses ..................... 51
  3.3.6 Recurrent Neural Network Mappings and Analyses ................... 56
  3.3.7 Convolutional Neural Network Mappings and Analyses ............. 62
  3.3.8 Tightly-Coupled Analog In-Memory Compute Tiles: Key Takeaways and Conclusions ......................................................... 65
3.4 Tightly-Coupled Systolic Arrays for Transformers ............ 66
  3.4.1 Systolic Arrays for Transformers .................................. 66
  3.4.2 Enabling TiC-SAT with ALPINE .................................... 67
  3.4.3 TiC-SAT Dataflow ................................................ 68
  3.4.4 TiC-SAT Experiments and Results .................................. 68
  3.4.5 Tightly-Coupled Systolic Arrays: Key Takeaways and Conclusions ........... 70
3.5 Cross-Solution Case Study: Comparing Caches, SIMD, Analog In-Memory Compute Tiles, and Systolic Arrays .......... 71
  3.5.1 Competing Solutions for Machine Learning Applications ............. 71
  3.5.2 Overview of Competing Edge AI Solutions ........................... 71
  3.5.3 Experimental Setup of Cross-Solution Study ......................... 73
  3.5.4 Cross-Solution Case Study Results and Analysis .................... 77
  3.5.5 Cross-Solution Case Study Takeaways and Conclusions .................. 79
3.6 Summary ............................................................. 79

4 Loosely-Coupled Accelerator Solutions 81
4.1 Introduction .......................................................... 81
4.2 Related Work ........................................................ 82
  4.2.1 Coupling of AIMC Tiles ............................................. 82
  4.2.2 Load Balancing Hardware Resources in Edge-AI Inference .......... 84
4.3 Mapping ML Kernels into Hybrid Analog-Digital Computing Layers .......... 84
  4.3.1 Hybrid Analog-Digital MVMs ....................................... 85
  4.3.2 Hybrid Layer Study Experimental Setup ............................. 86
  4.3.3 Dense Layer Study and Results .................................... 87
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>Chapter 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.4 Convolutional Layer Study and Results</td>
<td>89</td>
</tr>
<tr>
<td>4.3.5 Hybrid Layer Study Takeaways and Limitations</td>
<td>94</td>
</tr>
<tr>
<td>4.4 ALPINE Loosely-Coupled Framework</td>
<td>94</td>
</tr>
<tr>
<td>4.4.1 Tightly Coupled AIMC Tile Model</td>
<td>94</td>
</tr>
<tr>
<td>4.4.2 Modeling and Implementation of Loosely-Coupled AIMC Tiles</td>
<td>95</td>
</tr>
<tr>
<td>4.4.3 Loosely-Coupled AIMC Cluster Interface</td>
<td>96</td>
</tr>
<tr>
<td>4.4.4 Software Support: Expanding AIMClib with tiny tensorlib</td>
<td>96</td>
</tr>
<tr>
<td>4.5 Load Balancing Methodology</td>
<td>97</td>
</tr>
<tr>
<td>4.5.1 Load Balancing Approach and Objectives</td>
<td>97</td>
</tr>
<tr>
<td>4.5.2 Load Balancing Tightly Coupled AIMC Tiles and CPUs</td>
<td>99</td>
</tr>
<tr>
<td>4.5.3 Load Balancing Loosely Coupled AIMC Clusters and CPUs</td>
<td>100</td>
</tr>
<tr>
<td>4.6 ALPINE Load Balancing Experimental Setup</td>
<td>101</td>
</tr>
<tr>
<td>4.6.1 ALPINE Target Systems</td>
<td>101</td>
</tr>
<tr>
<td>4.6.2 Target Applications and Load Balancing</td>
<td>101</td>
</tr>
<tr>
<td>4.7 Tightly-Coupled versus Loosely-Coupled Results</td>
<td>102</td>
</tr>
<tr>
<td>4.7.1 Overall Results</td>
<td>102</td>
</tr>
<tr>
<td>4.7.2 MobileNetV2 Results and Analysis</td>
<td>104</td>
</tr>
<tr>
<td>4.7.3 VGG Results and Analysis</td>
<td>105</td>
</tr>
<tr>
<td>4.7.4 ResNet Results and Analysis</td>
<td>106</td>
</tr>
<tr>
<td>4.8 Summary</td>
<td>107</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>109</td>
</tr>
<tr>
<td>5.1 Summary of Contributions</td>
<td>109</td>
</tr>
<tr>
<td>5.1.1 gXR5 and ALPINE</td>
<td>109</td>
</tr>
<tr>
<td>5.1.2 Tightly-Coupled Edge AI Solutions</td>
<td>110</td>
</tr>
<tr>
<td>5.1.3 Loosely-Coupled Edge AI Solutions</td>
<td>111</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>112</td>
</tr>
<tr>
<td>5.2.1 Overcoming Communication Bottlenecks with Wireless Scratchpad Memories</td>
<td>112</td>
</tr>
<tr>
<td>5.2.2 Coherent Accelerators and Overcoming Address Translation Bottlenecks with Midgard</td>
<td>113</td>
</tr>
<tr>
<td>5.2.3 Validated Full System-Level GPU Modeling</td>
<td>113</td>
</tr>
<tr>
<td>Bibliography</td>
<td>126</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>127</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Trade-offs of different kinds of simulators, recreated and slightly modified from [41]. The Y-axis represents the accuracy of performance and run-time statistics of the simulated hardware with respect to real hardware (higher from the origin is better), and then the X-axis represents the quickness with which researchers can use, develop, and obtain statistics from these simulators (further right from the origin is better). 13

2.2 gXR5 Full System-Level model in gem5. All objects inherit from the system object during simulation run-time, hence the "system.x" notation, and are connected to other timing objects through various ports. Newly implemented models specifically for gXR5 include the Page Table Walker ("system.cpu.itb.walker.port" and "system.cpu.dtb.walker.port"), the SimpleBoard platform ("system.board"), the PLIC ("system.board.plic"), and the CLINT ("system.board.clint"). 17

2.3 gXR5 model hierarchy. 18

2.4 gXR5 Full System-Level architectural model, showing the relationship between the CPU core, SimpleBoard SoC, and the memory subsystem. 18

2.5 RISC-V Sv39 virtual address. 21

2.6 RISC-V Sv39 page table entry. 21

2.7 RISC-V Sv39 physical address. 21

2.8 Intensity-level classifications of microbenchmarks utilized to guide gXR5 validation. 28

2.9 gXR5 validation at the end of each set of iterations dedicated to validating functional units, branch predictor, memory, and pipeline models. 30

2.10 IPC of the HF1 board, pre-validation gXR5 model, and post-validation gXR5 model for each of the stress-ng benchmarks. 32

2.11 Execution time of the HF1 board, pre-validation gXR5 model, and post-validation gXR5 model for each of the SPEC CPU2017 benchmarks. The #XX indicates the percent of load/store instructions within each benchmark. 32

2.12 ALPINE Wrapper Object. Input control and output can refer to multiple kinds of interfaces, including DMA, IO bus, and ISA extension, and are meant to orchestrate accelerator functions. 33

2.13 Library structure of tinytensorlib and AIMClib with respect to a NN inference application. 34
3.1 Analog in-memory acceleration for neural network inference. (a) Phase-change memory (PCM) device with true analog storage capability (32 representative levels shown on real hardware experimentally). (b) Matrix-vector multiplication with PCM devices. (c) Analog in-memory computing (AIMC) tile with PCM array, data converters and digital control unit. (d) Weights of MLPs, RNNs, and CNNs represented with AIMC tiles. ................................. 41

3.2 A high-level overview of how a tightly-coupled AIMC tile could interact with an ARM Cortex-A55 pipeline. A dedicated line from an AIMC control pipeline can send signals and data directly to and from the AIMC tile's control unit, akin to a co-processor. .................................................. 43

3.3 (a) Visualization of the AIMC tile data-flow when using the ALPINE ISA extension. (b) The instruction definitions. To perform one matrix vector multiplication ($y = M \ast x$), first $CM\_QUEUE$ is called to send $x$ to the AIMC tile's input memory. Next, $CM\_PROCESS$ is called to perform $y = M \ast x$, which takes the $x$ sitting in the AIMC tile's input memory, performs multiply-and-accumulate operations on each of the crossbar columns, and then stores the output $y$ in the AIMC core's output memory. Finally, $CM\_DEQUEUE$ is used to send $y$ back to the CPU. ................................. 44

3.4 A sample C++ code for a single fully-connected feed-forward layer programmed onto AIMC tiles with AIMClib. At each inference step, the input is loaded and queued into the input memory. This is followed by the MVM via the aimcProcess method. Finally, the contents of the AIMC core output memory are dequeued straight into an output matrix data structure. ................................. 45

3.5 Software view of the AIMC object and AIMC tile object implementation in gem5-X: The AIMC object controls the generation of AIMC tiles (how many, sizes $M$ and $N$), while the tiles contain parameterizable crossbars and associated input and output memories. A variety of inputs can be used to control the function of the AIMC tiles which can be forwarded to various outputs (e.g., I/O Bus, CPUs). .................. 46

3.6 gem5-X Configuration Script code example for placing the AIMC wrapper class on the ARM SoC (RealView) platform. The instantiation is done in a similar manner for the loose-coupled and tight-coupled integration. ................................. 47

3.7 (a) Multi-layer perceptron architecture with two dense (fully-connected) layers (1024, 1024) and ReLU activation. (b) Cases 1 through 4 show the mappings of the full-connected layers to a variety of AIMC-based configurations. ................................. 51

3.8 Aggregate results for multi-layer perceptron experiments. From left to right, each column contains total time, memory intensity, and energy results for the High-Power system (top row) and Low-Power system (bottom row) configurations. "ANA" refers to analog AIMC-enabled application mappings with implementation numbers corresponding to those in Fig. 3.7, while "DIG" refers to a digital reference or CPU-only implementation. Results are also grouped by the number of CPU cores utilized (1, 2, or 4). ................................. 52
3.9 The run-time analysis of MLP cases. Run-time percentage of each sub-ROI division in MLP cases averaged across high-power and low-power systems. The average reference is obtained across single-, dual-, and quad-core digital cases. Non-MVM Digital Operations refers to the combined run times of the input load, digital activation, and output writeback. The standard deviation of the timing distribution is less than 1.7%, 1.2%, 2.3%, and 1.5% for the average reference, Case 1, Case 3, and Case 4, respectively. The time distribution of cases 1 and 2 are similar and hence, Case 2 is omitted for brevity. ........................................... 53

3.10 (a) The neural network diagram for the LSTM modeled in our AIMC-enabled test programs. It is a 2-layer LSTM with one LSTM hidden cell layer and one dense (fully-connected) layer with various, more compute-intensive, activation functions. Note that the AIMC sizes are variable depending on the size of the hidden cell layer and that all activation functions are performed in the CPU cores. (b) Cases 1 through 4 show the mappings of the layers to a variety of AIMC-based configurations. ........................................... 56

3.11 Aggregate results for all LSTM experiments. From left to right, each column contains total time, memory intensity, and energy results for the high-power system (top row) and low-power system (bottom row) configurations. "DIG" refers to digital reference applications while "ANA" refers to a specific analog, AIMC-enabled application case, which correspond to those in Figure 3.10. Results are grouped by the number of CPU cores utilized, and from left to right, each grouped column refers to a different \( n_h \) parameter, which affects the total size of the network. The darker bars refer to a larger \( n_h \). ................................. 58

3.12 Bar graphs showing the percentage of the ROI run time going towards sub-ROIs for the analog cases 1 through 4 run on the high-power system configuration for all \( n_h \) values. The trends of the high-power system sub-ROI run times are followed by the same cases in the low-power system configuration. The digital cases (single-, dual-, and quin-CPU core) see 87.8% to 97.9% of their total ROI run time dedicated to the digital MVM operations with activation functions in the cell layer. The "Misc." run time is comprised of all other sub-ROIs per experiment that comprise of less than 5% of the total ROI run time. ................................. 58

3.13 (a) The architecture of the CNNs presented in [109] and their mapping onto the ALPINE systems. The blue boxes with the AIMC tiles represent the convolutional layers. The dense layers are not mapped to AIMC tiles. (b) shows the dimensions and parameters of each CNN. The CNN has 5 convolutional layers (3 with Max Pooling), 3 dense layers, and ReLU activation functions for all layers except the last layer, which uses Softmax. ........................................... 62
3.14 Aggregate results for CNN experiments. From left to right, each column contains total time, memory intensity, and energy results for the High-Power system (top row) and Low-Power system (bottom row) configurations. ANA refers to analog AIMC-enabled applications with CNN names corresponding to those in Table 3.13, while DIG refers to a digital reference, non-AIMC-enabled, implementation. The CNNs F, M, and S represent fast, medium, and slow variations on the same CNN architecture, respectively. 

3.15 CPU utilization for CNN-S in the high-power system, expressed as the percentage of idle CPU cycles (top) and instructions/cycle (IPC) (bottom).

3.16 A high-level overview of a systolic array performing a matrix-matrix multiplication operation, recreated from Figure 2 left in [40]. The 3x4 yellow Input matrix is loaded into the systolic array, one element in each row every clock cycle in a staggered fashion. Weights are held stationary in the blue systolic array DPUs. The 4x3 green output matrix is obtained in a staggered format after less than 15 clock cycles.

3.17 A 2x2 systolic array of weight-stationary DPUs, with the DPU model being recreated from Figure 2 right of [40]. The weights are programmed in each green cell before application run time. The inputs are propagated column-wise from a two element-tall matrix and an accumulator is propagated row-wise from a zero matrix during run time.

3.18 (a) The architecture of the CNNs presented in [109] and their mapping onto the ALPINE systems. The blue boxes with the AIMC tiles represent the convolutional layers. The dense layers are not mapped to AIMC tiles. (b) shows the dimensions and parameters of each CNN. The CNN has five convolutional layers (three with Max Pooling), three dense layers, and ReLU activation functions for all layers except the last layer, which uses Softmax.

3.19 Matrix of results organized row-wise by metric (run time, energy, and L1 cache miss count per inference) for SIMD-enabled, AIMC-enabled, more-caches (16MB L2 and 32MB L2), and systolic array (TiC-SAT) systems. The results are arranged column-wise by Chatfield VGG8 variant (F(ast), M(edium), and S(low), for left, middle, and right columns, respectively). The stacked bars in the L1 cache miss graphs show the per-CPU cache misses. Note that the TiC-SAT energy does not include the accelerator energy, as there was no power model available in [120].

4.1 The ways in which layers too large for an AIMC tile can be mapped and how subsequent computation is divided for the equation, \(\text{Output} = A \ast X\), where \(A\) is the matrix of weights and \(X\) is the input vector. The portion of the weights matrix \(A\) that can be placed in the analog (ANA) are shown using the crossbar figure (light blue boxes). (a) shows the case where all weights can fit in the accelerator. (b) and (c) show cases where the accelerator can cover the width and height of the weights matrix, respectively, but not both. (d) shows a piece-wise case where the accelerator can cover neither the width nor height of the weights matrix.
4.2 Dense layer study graph showing factor speedup over digital reference versus layer coverage by AIMC tiles. Each point represents one experiment where 10 inferences are run on a single layer with some percentage of AIMC tile coverage. The red dotted line indicates 1x speedup over digital reference (analog and digital run times are equal). .................................................. 88

4.3 Dense layer study graph showing the number of DIG MACs versus the run time of the dense layer. The "leftover" DIG MACs are calculated by first computing the total DIG MACs, and then subtracting the number of MACs that are performed by the AIMC tiles, given a certain percentage of coverage. Each point represents one experiment where 10 inferences are run on a single layer with some percentage of AIMC tile coverage. In general, further right on the graph is less AIMC tile coverage of the layer. .................................................. 89

4.4 Plot showing the im2col transformations for all convolutional layers in MobileNetV2, ResNet50v1.5, and VGG16 [65], [144], [145]. The size of the markers correlates to the number of layers that have the same size. .................................................. 90

4.5 Convolutional layer study graph showing factor speedup over digital reference versus layer coverage by AIMC tiles. Each point represents one experiment where 10 inferences are run on a single layer with some percentage of AIMC tile coverage. The red dotted line indicates 1x speedup over digital reference (analog and digital run times are equal). .................................................. 91

4.6 Convolutional layer study graph showing the number of DIG MACs versus the run time of the dense layer. The "total" DIG MACs are calculated by first computing the total DIG MACs, and then subtracting the number of MACs that are performed by the AIMC tiles, given a certain percentage of coverage. Each point represents one experiment where 10 inferences are run on a single layer with some percentage of AIMC tile coverage. In general, further right on the graph is less AIMC tile coverage of the layer. Points with for the same layer with the same run time but different total DIG MACs value show the difference between height-wise coverage and width-wise coverage by the AIMC tiles. ............. 92

4.7 Block diagram of loosely-coupled ALPINE system model. The IO bus acts as the main interface between CPUs and the AIMC cluster, which is made up of multiple AIMC tiles and a controller. .................................................. 95

4.8 A diagram of tinytensorlib's placement and features relative to AIMClib and a NN application. tinytensorlib can inherit from AIMClib to incorporate tightly-and loosely-coupled AIMC interfaces, but can also exclude AIMClib for use with digital-only applications. .................................................. 97

4.9 Figure showing the steps of the reference and proposed load balancing algorithms, "reference" (REF) in (a) and "Prioritize Largest Layer" (PLL) in (b). The graphs show a simple NN with two convolutional layers in yellow boxes in a series of graphs, where the Y-axis is the number of DIG MACs for each of the two layers. At each step of each algorithm, one AIMC tile (blue boxes) is assigned. ................. 98
4.10 Pseudo-code for the reference (REF) load balancing approach for tightly-coupled systems. This is a "top-down" approach where first layers are assigned to CPUs, and then subject to the number of tightly-coupled AIMC tiles per CPU, tiles are assigned to layers. ................................................................. 99

4.11 Pseudo-code for a reference load balancing algorithm for loosely-coupled systems. This is a "bottom-up" approach where tiles are allocated to layers first, and then layers are assigned to CPUs. To assign a tile to the layer with most DIG MACs, I simply iterate over the layers of the NN and calculates the DIG MACs for each layer. ................................................................. 100

4.12 Figure showing time (s) per inference (row 0), speedup factors (row 1), and energy improvement factors (row 2) for all tested NNs using tightly- (TC) and loosely-coupled (LC) systems with In-Order CPU models. The gray line shows the SIMD-enabled, non-AIMC tile-enabled baseline that all results are compared to. The X-axis organized by the number of tiles in the system, e.g., TC2 is two tightly-coupled AIMC tiles per CPU core, which given 8 CPU cores in the system, is 16 AIMC tiles total, and LC64 is 64 loosely-coupled AIMC tiles for the whole system. REF refers to the reference load balancing algorithm while PLL refers to the prioritize largest layer load balancing algorithm. ................................. 103
# List of Tables

2.1 Instructions introduced in the RISC-V Privileged ISA specification and their bit layouts. ........................................ 19
2.2 Missing fence instruction from the RV32/64 Base Instruction Set. .................. 20
2.3 Missing fence instruction from the RV32/RV64 Zifencei Standard Extension. .... 20
2.4 Simulator Model and HiFive Unleashed Technical Specifications ................. 26
2.5 Selected Benchmarks for gXR5 Validation ([62], [63]) ............................... 28
2.6 gXR5 Validation Parameter Values ............................................. 31

3.1 ALPINE Tightly-Coupled AIMC Tiles Experimental Setup ......................... 49
3.2 Aggregate MLP Results .......................................................... 52
3.3 LSTM Experiment Setup ................................................................ 57
3.4 Aggregate LSTM Results ................................................................ 59
3.5 Aggregate CNN Results ............................................................. 64
3.6 BERT-large Run Time Results ....................................................... 69
3.7 BERT-large sub-ROI Run Time Results ........................................... 69
3.8 Transformer Results .................................................................. 70
3.9 System Configurations ............................................................... 73
3.10 Chatfield CNNs Layer Dimensions ................................................ 75
3.11 Chatfield CNNs AIMC Tile Coverage ............................................. 76

4.1 Tight versus Loose Coupling of Accelerators ............................................ 83
4.2 Hybrid Layer Study System Specifications .............................................. 87
4.3 Hybrid Layer Study: Selected Convolutional Layer Parameters ................. 91
4.4 Convolutional Layer Speedups and MACs per Data Transfer (100% Layer Coverage) .............................. 93
4.5 System and AIMC Energy Model ...................................................... 101
4.6 Tested CNN Parameters .................................................................. 102
4.7 Tested CNN Statistics .................................................................... 102
4.8 Maximum Speedup and Energy Factors ................................................. 103
Acronyms

AI  Artificial Intelligence
AIMC  Analog In-Memory Computing
ARM  Advanced RISC Machine
CNN  Convolutional Neural Network
CPU  Central Processing Unit
DL  Deep Learning
IoT  Internet-of-Things
ISA  Instruction Set Architecture
LLM  Large Language Model
ML  Machine Learning
MLP  Multi-layer Perceptron
MMIO  Memory-Mapped Input-Output
MMM  Matrix-Matrix Multiplication
MPSoC  Multiprocessor Systems-on-Chip
MVM  Matrix-Vector Multiplication
NN  Neural Network
OS  Operating System
PCI  Peripheral Component Interconnect
PCIe  Peripheral Component Interconnect Express
PIO  Peripheral Input-Output Device
RISC  Reduced Instruction Set Computer
RNN  Recurrent Neural Network
ROI  Region of Interest
SIMD  Single Instruction Multiple Data
TiC-SAT  Tightly-Coupled Small-Scale Systolic Array
1 Introduction

1.1 AI and Edge Systems

Due to the emergence of big data, the field of Artificial Intelligence (AI) is currently undergoing a Renaissance. In 2022, Large Language Model (LLM) networks such as Chat-GPT and image synthesis models like Midjourney captured the world’s attention with their ability to synthesize and create human-like text and realistic images, respectively [1], [2]. These models can inspire artists, streamline search, assist programmers, automate complex tasks, synthesize data, and a lot more [3]–[5]. The same pressure that leads to large and innovative Neural Network (NN) models is also pushing AI to the edge, to enable highly distributed and complex tasks such as biodiversity, pollution, and healthcare monitoring [6]–[8].

Due to performance bottlenecks, security concerns, and reliability issues in long-distance network communications, a lot of tasks performed at the edge cannot rely on processing by a centralized server [9]. As a result, the capabilities and performance of edge devices have expanded in recent years with advances in computer system design, to the point where running AI inference is now feasible on a variety of edge devices [10]. These edge devices, which typically employ Reduced Instruction Set Computer (RISC) architectures such as Advanced RISC Machine (ARM) and RISC-V, are known to prioritize energy efficiency and form factor (chip area) while still delivering performance now that was on-par with what supercomputers could achieve decades ago.

But as energy efficient as edge devices may be, AI-enabled applications demand ever-increasing compute and memory resources, stressing their typical power-constrained allotment. AI, Machine Learning (ML), and Deep Learning (DL) applications are incredibly memory-intensive for the storage requirements of millions, if not billions, of weights and parameters, as well as compute-intensive due to the underlying matrix-vector and matrix-matrix algorithms that have exponential compute complexity [11]. These compute and memory intensities put AI-enabled applications at odds with the power-, compute-, and area-constrained nature of edge devices, and thus puts severe limitations on the size of neural networks that can be run on them.
Chapter 1 Introduction

To address the complexity and memory footprint of AI-enabled applications, the underlying NN is often decreased in size by either limiting hyperparameters or by reducing the precision of weights. However, this comes at significant cost to NN accuracy, sometimes rendering them unusable. Furthermore, over time, this is counter to a general trend of neural networks growth: in 2015, I. Goodfellow et al. showed that the number of neurons in modern neural networks approximately doubles every 2.4 years, when fitting state-of-the-art neural networks from 1958 onwards [12]. L. Bernstein et al. goes further to show the number of multiplication operations in neural networks increasing by an order of magnitude every approximately 18 months between 2014 and 2021 [13]; thus, the ever-increasing system requirements of NN is a challenge for the scalability of modern computer architectures. Therefore, as AI-enabled applications continue to grow, there is a real challenge to run modern and future neural networks. Otherwise, these networks will become infeasible at the edge. At its core, this risk is perpetuated by the power wall and the memory wall.

1.2 The Power and Memory Walls

The challenge to operating and maintaining current and future AI-enabled applications at the edge is due to architectural scalability. Without specific regard to AI computing and edge domains, computer architects have a long history of grappling with challenges to scalability in their quest to increase system performance. Fundamentally, the von Neumann paradigm of computing itself – the idea that data is loaded to compute from memory, and then stored back in memory – is what limits the capabilities of typical general-purpose edge devices, and therefore new paradigms of computing are needed [14].

More specifically, Moore's "law", the prophecy of doubling transistor density every 18 months, is reaching a physical upper bound due to quantum effects and complexities in fabrication, thus limiting compute power over time. This, combined with Dennard scaling – the approximation of constant power density for any transistor size due to leakage power – foresees a Central Processing Unit (CPU) having an unmaintainable power density as transistors shrink. These two trends combined create a so-called "Power Wall", where future computer architectures will not be able to attain higher core frequencies and transistor density as a means of achieving greater performance, because the energy cost to supply the necessary power and maintain an optimal temperature is too high [15]–[17].

To maintain growing performance and Moore's law, computer architects started leveraging parallelization: multiple CPU cores with their own cache subsystems, shared caches, task scheduling at the kernel level, and so on. However, because compute is now more distributed, a new wall emerges: the "Memory Wall" [18], [19]. In short, while compute performance doubles every 18 months with Moore's law, memory performance only seems to double every decade. The divergent trends in processing and memory performance predict that the difference in performance between compute and memory resources will become so high that any computation is necessarily bounded by data movement. Larger applications,
and especially AI-enabled applications, with increasing memory and compute requirements, perpetuate this memory wall.

Therefore, while we can speed up and make more energy efficient AI-enabled applications through the use of parallelization and Multiprocessor Systems-on-Chip (MPSoC), the distribution of more and varied compute resources exacerbates data movement-based bottlenecks. The power and memory walls combined mean that in order for edge devices to run modern and future AI-enabled applications, general-purpose homogeneous architectures will not be enough. Therefore, new heterogeneous architectures – architectures that change the von Neumann design – and methodologies must be explored.

1.3 Exploring Heterogeneous AI Solutions

Of course, the aforementioned issues with power and memory walls have been known by computer architects for several decades. As a result, to continue to meet the power, compute, and area constraints of edge devices, computer architects and AI specialists have already started prototyping, validating, and implementing numerous heterogeneous architectures. Modern edge AI systems are constructed using small general-purpose systems with additional specialized and energy-efficient hardware, including but not limited to, neural network engines, in-memory compute accelerators, Single Instruction Multiple Data (SIMD) co-processors, and more [20]–[23].

The process of building a new heterogeneous edge device for AI-enabled applications, however, is only increasing in complexity. The most popular software tools and frameworks for running AI-enabled applications, such as TensorFlow and PyTorch, typically require an Operating System (OS) and OS-level libraries to run their applications [24], [25]. Software engineers and AI specialists want to assume a functioning and easy-to-use underlying software stack when they build their NN models. In contrast, hardware designers working on accelerators tend to focus on specialized hardware for specialized neural networks, with a narrow test and validate scope. Simulating, testing, and maintaining a full-fledged software stack is often out of the scope and impractical to run in circuit-level simulators, due to extreme simulation development and run time overheads [26].

Therefore, it is necessary to develop the computer architecture frameworks and simulation infrastructures that link the high-level performance and behaviour of AI-enabled applications to the interactions of low-level hardware in such a way that computer architects can quickly explore the benefits and bottlenecks of future heterogeneous architectures. Irrespective of AI-enabled applications, this is the argument for gem5 and gem5-X: a full system-level computer architecture simulator that simulates major hardware components and routines as tunable black boxes in software, and thus can run high-level applications in user space atop an OS and disk image while producing performance statistics representative of real hardware. Instead of a circuit-based timings simulation that must maintain the state of hardware, wires, netlists, etc., gem5 and its variants are event-based cycle-accurate simulators that maintain a queue of
Chapter 1 Introduction

timed events to preserve high-level behaviour of the underlying hardware [27].

gem5 is modular and generic enough that it is routinely extended by researchers to simulate a wide variety of systems, including heterogeneous systems from the edge domain [27]. gem5-X then extends gem5 by providing out-of-the-box validated system models that ensure experiments produce statistics representative of real ARM hardware, as well as a methodology for conducting efficient architectural explorations [28].

Despite its modularity though, when gem5 (or other full system-level simulators) are extended to explore new architectures, there is typically a very limited scope in mind. Often, when a simulation infrastructure is released in the literature, it is meant to simulate "state-of-the-art" – homogeneous general-purpose architectures – against a "novel" architecture, or the solution being proposed; and critically, only that solution being proposed [23], [29]–[32]. Thus, when faced with so many competing heterogeneous architecture solutions, it is difficult to impossible to weigh the trade-offs in implementation, usability, performance, energy gains, and so on, of these different solutions.

The goal of this thesis is to extend and specialize the gem5 simulator towards simulation of many heterogeneous edge architectures by expanding upon the dimensions of which heterogeneous architecture exploration can be conducted. Using AI-enabled applications as the target case then, I explore different forms of heterogeneity, including in the form of different accelerator architectures, different interfaces, and different communication paradigms.

1.4 Thesis Contributions

Given our knowledge of AI-enabled applications’ compute complexity, large data footprint, and the memory wall, we can assert three simultaneous bottlenecks that arise in AI-enabled applications: communication, memory, and compute. These bottlenecks must be addressed simultaneously to progress state-of-the-art edge architectures, and therefore computer architects targeting edge AI must have tools that enable architectural explorations of these bottlenecks. Fundamentally, then, computer architects need full system-level tools to explore not only different accelerators in heterogeneous systems, but also their means of integration and their interfaces. Therefore, in this thesis, I introduce the ALPINE framework: methodologies and tools for exploring heterogeneous systems targeting the operation of AI-enabled applications. A list of contributions is as follows:

- To facilitate architectural explorations, unhindered by barriers to instruction set architecture use, I developed the world’s first Linux-capable full system level simulator for architectures based on the open and free RISC-V ISA, gXR5. gXR5 is capable of running full system-level simulations of RISC-V systems and is validated against selected stressing and SPEC CPU2017 benchmarks to produce performance statistics representative of real RISC-V hardware, with a sub-20% mean error.
• To target explorations of novel edge AI accelerators and their interfaces, a new generic peripheral input-output device model is implemented as the basis for ALPINE. This model facilitates computer architects in their ability to implement and conduct architectural explorations with novel edge AI accelerators by allowing easy fine-tuning and implementation of numerous state-of-the-art accelerators and their interfaces.

• As a preliminary showcase of ALPINE, I implement and validate Analog In-Memory Computing (AIMC) tiles close to the CPU and interfaced via ISA extension in a so-called "tightly-coupled" configuration.

• Using ALPINE, I then perform numerous case studies of applying AIMC tiles to Multi-layer Perceptrons (MLPs), Recurrent Neural Networks (RNNs), and Convolutional Neural Networks (CNNs), to show the usability and applicability of this AI solution as well as ultimately up to 20.5x speedup and 20.8x energy gains over conventional SIMD-vectorized systems.

• This ALPINE accelerator model is also used to explore Tightly-Coupled Small-Scale Systolic Arrays (TiC-SATs), where, using Transformers as an initial test case, we show up to 89.5x application-wide speed-ups across Transformer models.

• With multiple accelerator models in ALPINE, I am able to compare directly competing AI solutions in CNNs by weighing the trade-offs of capabilities, speedups, and energy gains of two accelerator-based solutions in the same system, and also against systems with a SIMD vectorized co-processor and larger caches.

• To increase the dimensions of heterogeneity in architectural explorations, I perform additional studies of the AIMC tiles that are placed far away from the CPU with a bus-based data transfer interface – a so-called "loosely-coupled" interface – and compare performance statistics directly with tightly-coupled AIMC solutions. This exploration reveals the role of load-balancing and data transfer overheads in modern CNNs, and thus by tuning state-of-the-art load-balancing algorithms, we can achieve additional speedups and energy gains (up to 100%) in a wide variety of convolutional neural networks.

• To ensure the future viability of the ALPINE framework, I assist in the exploration of heterogeneous architecture solutions that target communications bottlenecks with wireless scratchpads for cross-core communication. Compared to only a system with tightly-coupled AIMC tiles, an additional 20% speedup and energy gains can be achieved by addressing communication bottlenecks.

In this section, I briefly discuss the contributions of each of the following chapters.
1.4.1 Apparatuses and Tools: gXR5 and ALPINE Frameworks

**gXR5: Linux-Capable Full System-Level of RISC-V Systems**

To conduct full system-level explorations of heterogeneous edge AI architectures, it is necessary to be able to model RISC Instruction Set Architectures (ISAs). This is because RISC ISAs are known to supply highly energy efficient edge, mobile, and Internet-of-Things (IoT)-domain devices [33]. At the time of development, full system-level simulation of Linux-capable RISC systems that were validated against real hardware were limited to these ARM ISA-based systems [28]. However, using the ARM ISA for real systems, including architectural explorations, is hindered by prohibitively expensive licensing fees, which propagate closed-source hardware models [34].

To overcome this limitation, the RISC-V ISA was introduced as a free and open-source ISA. Following from the introduction of RISC-V, and to ensure the longevity of edge AI architectural explorations and facilitate open-source hardware, I implemented gem5 extensions for RISC-V, or gXR5: the world’s first Linux-capable full system-level computer architecture simulator for free open-source RISC-V ISA-based systems [35].

The effort to enable Linux-capable full system-level RISC-V simulation involved developing and implementing numerous hardware models for the RISC-V ISA, including MMU, interrupters, and timers. The operation of these models were then verified by implementing the full RISC-V software stack for simulation, including bootloader, Linux kernel, and disk image. Once operational, I contributed to the validation effort of gXR5 against the SiFive HiFive Unleashed SoC. Using a tuning methodology starting with microbenchmarks from the stress-ng suite, and then further tuning gXR5 against select SPEC CPU2017 integer benchmarks, we attain performance statistics from gXR5 with a mean error of less than 24% with respect to real hardware [36].

This validation effort has been published in the 2023 RISC-V Summit in Europe [36].

**ALPINE: Enabling Architectural Explorations of Heterogeneous Edge AI Solutions**

In order to simulate heterogeneous edge AI architectures, new models in full system-level simulation must be introduced that can encapsulate general accelerator timings, behaviour, and interfaces. To that end, I model within the ALPINE framework a generic Peripheral Input-Output Device (PIO) device model that can be easily tuned to simulate accelerator functions while also maintaining multiple different interfaces for multiple system configurations.

The ALPINE generic PIO model is modeled as an on-chip device akin to system peripherals, but it includes both connections to the system IO bus and a direct line to CPUs. The direct line is interfaced within CPUs to act as a so-called “tightly-coupled” interface, where the PIO model can be accessed quickly via ISA extension and tuned via instruction operation latencies. In principle, this models a system where CPUs have an additional ALU pipeline dedicated to
accelerator operation, and this particular pipeline interfaces the controller of the accelerator directly for orchestrating data transfers and accelerator compute operations. In contrast, the connection to the IO bus enables more traditional long-distance interfaces, so-called "loosely-coupled" interfaces, such as interface via Memory-Mapped Input-Output (MMIO) and/or kernel driver. The loosely-coupled interface can utilize on-system DMA controllers to beam data to and from the accelerator to different locations within the system.

Having both interfaces in place means we can perform architectural explorations that weigh the trade-offs of performance, communications, and proximity. The loosely-coupled interface can, of course, be extended to more complex loosely-coupled interfaces such as Peripheral Component Interconnect (PCI) and Peripheral Component Interconnect Express (PCIe) [37]. The timings of the loosely-coupled module can be made as simple or complex as necessary, depending on tuned bus interaction response times.

With the interfaces set, the generic PIO model acts as a delegate for accelerator operation. In other words, given the tight or loose interface, this generic PIO model is capable of modeling a wide range of accelerators. In later sections, we demonstrate the ALPINE PIO model's capabilities by implementing two different accelerators: systolic arrays and AIMC tiles.

### 1.4.2 Tightly-Coupled AI Accelerator Solutions

**Analog In-Memory Compute Tiles**

As an initial exploration into the usability of ALPINE to explore edge AI devices, I first model tightly-coupled AIMC tiles based on fabricated models from IBM Research, Zürich [38]. These in-memory computing accelerators target one of the primary bottlenecks in running NNs: Matrix-Vector Multiplications (MVMs). By leveraging low-power analog technologies and highly parallel multiply-and-accumulate operations in phase-change memories, these AIMC tiles can perform a MVM operation in constant time ($O(1)$), as opposed to the usual $O(n^2)$ complexity.

After modeling the AIMC tiles, I then implement AIMClib: a software library to easily interface AIMC tiles and integrate AIMC tiles into modern NNs. With both the validated hardware and the software interface set, I am able to conduct three preliminary exploration studies of modern AI-enabled applications – MLPs, RNNs, and CNNs – in multiple system configurations. These exploration studies look at multiple cases with different AIMC tile mappings for different layers of these networks, strategies to reduce overheads, and isolation of bottlenecks.

In the case of MLPs, we show over 8x speedup and energy gains over a reference SIMD-enabled system. The MLP study shows cross-core communication as an emerging bottleneck in edge AI applications. In the next case study, we show over 9x speedup and energy improvement in RNNs as well as prove the scalability of performance and energy with larger networks. In the last case study, we show up to over 20x speedup and energy gains in modern CNNs and...
reveal the role of load balancing in accelerated networks. All of these studies are augmented with complexity and space analyses which further quantify the scalability of neural networks utilizing AIMC tiles.

This work was published in IEEE Transactions on Computers (TC), 2022 [39].

**Systolic Arrays for Transformers**

To show the flexibility of ALPINE’s generic PIO model, I contributed to the effort to refactor the AIMC tile model to instead model a competing edge solution: systolic arrays. These TiC-SAT accelerators are optimized for Matrix-Matrix Multiplications (MMMs) with a fan-out-fan-in structure enabling parallel computation.

By placing systolic arrays close to the CPUs, the typically high data transfer cost associated with standard scratchpad-based interfaces is alleviated. Therefore, by applying TiC-SATs and data re-use strategies to modern Transformer applications, up to 89.5x speedups are achieved for the BERT-large transformer with a 16 by 16 TiC-SAT.

This work was published in the Asia and South Pacific Design Automation Conference (ASP-DAC), 2023 [40].

**Comparison of Multiple Tightly-Coupled Edge AI Solutions**

With two competing edge AI solutions implemented in ALPINE – AIMC tiles and TiC-SATs – it is now possible to directly compare these same edge AI solutions directly on common applications and systems, as well as to systems with SIMD and larger caches. In a preliminary study, we run CNN inference on all of these systems to show the complexities in mapping and utilizing different accelerators with the tightly-coupled paradigm, as well as compare the speedups, energy improvements, and bottlenecks of highly constrained systems.

Given a constant area allocation per CPU core for the accelerators or for more caches, we show the role of accelerator utilization in VGG8 CNN variants, how to reconfigure AIMC tiles in a heterogeneous system for more optimal performance, exploit TiC-SATs for convolutions, and how this compares to convention SIMD and cache-based solutions. Ultimately, at the cost of flexibility, small AIMC tiles attain over 2x speedup and energy gains over SIMD units and TiC-SATs occupying the same chip area, and over 9x speedup and energy gains for augmented cache systems.
1.4.3 Loosely-Coupled AI Accelerator Solutions

ALPINE-LC and Analog In-Memory Compute Tile Clusters

A vital trade-off in accelerator integration within heterogeneous systems is the trade-off of proximity to the CPU; with ALPINE tightly-coupled systems, we assume a close proximity of accelerators that enable the CPUs to handle additional logic easily and with high data throughput thanks to an ISA extension-based interface. Using ALPINE Loosely Coupled, or ALPINE-LC, I show how to configure, integrate, and interface AIMC tile clusters in a loosely-coupled fashion, building upon the tightly-coupled AIMC software library, AIMClib, to create tinytensorlib: an Eigen C++-backed minimalist library for quickly implementing highly optimized AIMC tile-enabled applications both in tightly-coupled and loosely-coupled configurations.

Which coupled is best coupled?

With both tightly-coupled and loosely-coupled interfaces implemented in ALPINE, I am able to directly compare tight and loose interfaces to AIMC tiles. I performed a large study of how to extract performance and energy benefits from edge AI AIMC solutions, given the trade-offs in network size, available AIMC tile resources, and configurations of AIMC tiles and clusters. I performed an initial single-layer study of hybrid digital-analog layers which showed that there exists a threshold from where we can attain benefits from AIMC tiles. We use this study to develop a heuristic metric – MACs per transfer rate – to guide and explain the performance behaviour of different AIMC tile allocations. We then explore the performance and energy benefits of two different load balancing techniques – distributed and priority for largest layers – for both CPUs and AIMC tiles in multi-core systems. Using five real edge CNNs, classed as MobileNets, ResNets, and classical CNNs, our analysis shows that tightly-coupled AIMC tiles, despite likely being more limited in resources, are better for neural networks that exhibit low data reuse and therefore a low MACs per transfer rate due to their very low data transfer overhead, while loosely-coupled AIMC clusters are better for classical CNNs due to their ability to more freely allocate more AIMC tiles to larger layers in order to meet thresholds that enable great performance gains. We ultimately show up to 5.9x speedup and 5.6x energy gains even in cases where AIMC tiles cover a small portion of the neural network.

This work has been submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS).
2 Apparatuses and Tools: gXR5 and ALPINE

2.1 Introduction

Even as the capabilities of edge devices have grown significantly in recent years – going from discrete chips that act as housing for only a few sensors and limited processing to send data to a centralized server, to being, for example, full-fledged mobile computer systems that employ operating systems, full memory hierarchies, multiple CPU cores, and more – edge devices remain constrained by the need to be energy efficient and cool in order to fulfill numerous functions. As AI-enabled applications emerge as a significant proportion of processing at the edge, this need to remain energy efficient and cool is stressed by the need to also be performant. As a result, edge devices regularly employ accelerators and other heterogeneous computing aspects to continue to meet time- and energy-based constraints.

The need to optimize performance and energy of edge devices therefore necessitates a full system-level simulator, capable of simulating both the full software architecture stack present in modern heterogeneous edge AI systems and the models of the underlying hardware, across multiple dimensions of heterogeneity where known AI-based bottlenecks occur. Furthermore, full system-level architectural explorations of mixed general-purpose AI-accelerator systems must be able to be conducted, unfettered by ISA licensing and closed-source hardware.

Given the compute and data-intensive nature of AI applications, we can already assert that the bottlenecks occur during CPU or accelerator compute, cross-chip (i.e., core to core, core to accelerator) communication, and via data movement in and out of the memory hierarchy. Several full system-level frameworks exist that can simulate full software stacks, validated hardware models, accelerators, their interfaces, and interactions with the memory hierarchy, but none of them can do all of this simultaneously and with the ability to easily extend heterogeneity to new accelerators and compute paradigms.

In this chapter, I present gem5 Extensions for RISC-V, gXR5, and the ALPINE framework. gXR5 is the world’s first Linux-capable full system-level simulator of open and free RISC-V systems, validated against real RISC-V hardware. gXR5 builds atop the highly modular gem5
architectural simulator to free architectural explorations of novel hardware from licensing
constraints and spearhead novel explorations of the most state-of-the-art hardware. With gXR5
providing RISC-V ISA support and predecessor gem5-X providing tools and methodologies for
quick heterogeneous architectural explorations of ARM ISA systems, I am then able to build the
ALPINE framework: a framework that enables full system-level architectural explorations of
tightly-coupled and loosely-coupled AI accelerators, with multiple interfaces. ALPINE targets
the dimensions of heterogeneity that will speed up and make more energy efficient modern
and future AI-enabled applications, by enabling numerous opportunities for optimization at
both the software and hardware level. In particular, the following contributions made by both
gXR5 and ALPINE are presented in this chapter:

- The gXR5 framework, which extends gem5 to be able to simulate 64-bit RISC-V systems
  running on in-order CPUs and a Linux OS.
- The validation effort and methodology to tune gXR5 to produce performance statistics
  representative of real RISC-V hardware, which yields sub-20% validation error across a
  variety of microbenchmarks from stress-ng and benchmarks from SPEC CPU2017.
- The ALPINE framework and PIO model that enables multiple edge-AI accelerator models
  and integration methodologies.
- The software-level frameworks AIMClib and tinytensorlib, which enable the ability to
  very quickly create and validate the performance of edge AI applications.

This chapter is organized as follows: First, in 2.2 I describe gXR5 and ALPINE’s direct predeces-
sor gem5-X, as well as related work and other frameworks and their limitations that necessitate
ALPINE and gXR5’s development. In 2.3, I then go into detail on gXR5’s implementation, the
features necessary to enable full system-level RISC-V simulations, and the validation effort to
ensure gXR5’s performance statistics are representative of real hardware. 2.4 then explains
the ALPINE framework and the underlying PIO model that enables heterogeneous system
explorations of edge AI accelerators. Finally, to easily leverage the ALPINE framework from the
application level, 2.4.2 goes over the implementation of ALPINE’s associated libraries: AIMClib
and tinytensorlib.

2.2 Background: Full system-level simulation, gem5-X, and Related
Work

2.2.1 Simulation Types and Trade-Offs

It is well known that the time-to-market of novel computer hardware is long, usually on the
order of several months to years. In order for computer hardware companies to keep up with
both growing demand and their competitors, these companies try to employ many techniques
to reduce the time-to-market of hardware, including using simulators to prototype and predict the behaviour of hardware before it is fabricated.

Broadly speaking, there are three categories of computer architecture simulator: hardware-level, functional-level, and full system-level. The trade-offs of these simulators, shown graphically in Figure 2.1, are primarily based off of resolution, i.e., the accuracy of statistics generated by the simulator with respect to real hardware, and the overhead of using the simulator.

Hardware-level simulators include Hardware Description Language (HDL) or Register Transfer Level (RTL) simulators such as those built for modeling Verilog, SystemC, or Chisel. These simulators tend to model the propagation of signals through wires, logic gates, combinational blocks, and more. As a result, these simulators always offer the most accurate performance and energy statistics with respect to real, fabricated hardware. However, as modern computers continue to grow in complexity by using chips with billions of transistors, and therefore millions of logic gates, wires, and other components, simulating an entire system (with or without a full software stack), using HDL or RTL simulators comes with unbearably high simulation run-time and development time overhead [42], [43]. As a result, these simulators are regularly used to model smaller system components before computer architects move to a different apparatus.

On the other end of the spectrum are so-called "functional-level" simulators, such as QEMU or spike. These simulators typically forego modeling any hardware or hardware routines in great detail to instead model system instruction execution using software. As a result, these simulators can very quickly (usually on the order of seconds or minutes) execute compiled binaries atop simulated Linux-capable systems. However, because major hardware components
and routines are abstracted out of existence, it is difficult to obtain detailed performance and energy statistics representative of real hardware [44], [45].

The middle-ground for statistics accuracy and simulation speed is then Full System-Level simulators. These simulators simulate the major hardware components and routines, such as ALU functional units, caches, memories, interrupts, buses, and others, as software black boxes with tunable attributes. While not as accurate as RTL/HDL simulators, full system-level simulators can be validated against real hardware by tuning the black box attributes (response latencies, bit width, etc.) of various hardware components, thus producing performance statistics representative of real hardware at a fraction of the time and development overhead [27], [46], [47].

As previously discussed, modern edge AI-enabled applications tend to employ full hardware and software stacks, and therefore the target of this work is full system-level simulation. There are many full system-level simulators supported today. However, to target heterogeneous architecture explorations, we ideally want to fulfill the following criteria:

- The simulator must be modular enough to be able to easily develop and integrate novel accelerator models and their interfaces.
- The simulator can run systems with RISC ISAs.
- The simulator supports simulating user-space AI-enabled applications atop modern operating systems and simulated general-purpose systems.
- The simulator is validated against real hardware.

In the following section, I go over numerous prior candidates for the target apparatus.

2.2.2 Related Works: Simulators for Heterogeneous Architectural Explorations

With the criteria specified in the previous sections, the options for adequate full system-level simulators that can target novel edge AI systems quickly evaporate.

If we first search published simulators that target simulation of AI solutions, we notice that all of these simulators are too targeted and not modular enough to support heterogeneous edge AI architectural explorations outside the scope of their presented solution. For example, PUMA targets simulation of a memristor-based accelerator for ML inference. It proposes a pipelined architecture to perform MVMs in ML applications, such as MLPs, RNNs, and CNNs. To evaluate their proposed architecture, the authors integrate applications via a compiler extension and custom ISA, and then through a custom-built simulator called PUMAsim. PUMAsim, while claiming to contain functional and timing models, does not appear to be full system-level with support for RISC ISAs, nor is it validated [48].
Ottavi et al. [49] proposes and synthesizes a heterogeneous architecture where an analog in-memory compute tile sits within a cluster of RISC-V processors. They perform a design-space exploration using the bottleneck of a MobileNetV2 CNN where point-wise and depth-wise convolutions are placed within the accelerator. However, their synthesis approach, while extremely modular, falls under the category of RTL simulation and thus hinders quick exploration of alternative architectures and models. This is also the case for [50] and [51], which both target systolic array-based solutions for AI-enabled applications.

A better example is possibly PIMSim, which uses the full system-level gem5 computer architecture simulator as a back-end for simulating a heterogeneous system capable of processing-in-memory, a similar paradigm used by PUMA for AI-enabled applications [52]. gem5 stands out as a full system-level simulator that is highly modular (as evidenced by hundreds of extensions proposed and implemented over more than 15 years), supports multiple ISAs (x86, ARMv7, ARMv8, SPARC, ALPHA), and system models (CPUs, caches, memory, buses, peripherals) with their interfaces, can run user-space applications atop an operating system and disk image [27], [53]. However, not all of its components work out-of-the-box and it does not come pre-validated against real hardware. Furthermore, with PIMSim only targets one kind of processing-in-memory architecture that is not even specific to edge AI devices nor AI-enabled applications, neither is it validated.

It is worth noting as well that other general-purpose full system-level simulators exist, but come with a different set of shortcomings. For example, Sniper and zsim are two such simulators that employ parallelization to speed up simulation. While the parallelization make these very tempting options for our apparatus, both only support x86 CISC architectures and have not been supported in several years [46], [54]. Numerous other non-parallel simulators exist [47], [55], but they too come with limitations in the form of limited ISAs, limited number of system models, and lack of validation.

### 2.2.3 Predecessor gem5-X

Taking note of the aforementioned issues with gem5, the gem5-X framework resolves numerous issues with gem5 and its other extended frameworks. Recognizing the need for fast architectural explorations, gem5-X is an open-source framework and methodology, targeting the simulation of heterogeneous architectures by providing validated models and all the full system-level components (disk image, Linux kernel) right out of the box for the ARM ISA. Furthermore, it applies its methodology for fast architectural explorations to a variety of architectures and applications [28].

gem5-X is validated against real ARM hardware: the ARM Juno board. The in-order MinorCPU model is validated against the ARM A53 in-order core while the out-of-order DerivO3CPU model is validated against the ARM A57 out-of-order core. The simulated ARM system in gem5-X achieves a mean absolute error (MAE) below 4% with respect to real hardware. In addition to validated CPU models, gem5-X also provides numerous architectural extensions,
including support for scratchpad memories and a 3D-stacked HBM2 model.

Recognizing the need to run full system-level simulations of modern systems with a complete software stack across a variety of computing domains, gem5-X provides a Linux v4.4 kernel binary and Ubuntu 16.04 LTS disk image. The gem5-X manual goes into detail about how to maintain the disk image using the tool chroot and a static QEMU binary, so that the longevity of gem5-X can be maintained.

To assist with fast simulations, gem5-X also features numerous support enhancements, including in-simulator profiling with gperf, enhanced check-pointing, and 9P-over-VirtIO host-guest file sharing. With these enhancements, the file system and OS can be booted quickly using a simple atomic CPU model. After booting the main system and application, a checkpoint is taken such that the simulation can be rebooted using one of the validated CPU models and then timings-based simulation for validated performance statistics can be conducted.

The gem5-X manual also details numerous additional extensions for explorations of novel heterogeneous solutions. For example, the ISA extension in gem5-X was showcased in [29], where unused op-codes in the ARM ISA were used to implement new custom instructions that enable a binary dot product engine for binary neural networks. This same ISA extension was also used for BLADE in-cache computing to verify the performance of Kvazaar video encoding [56], [57].

In summary, gem5-X fulfills our criteria as a starting point for heterogeneous architecture explorations for AI-enabled applications as it is based on the modular gem5 framework, supports the ARM ISA which is a RISC type, supports running user-space applications atop a modern OS and filesystem, and is validated against real ARM hardware. The roadblocks left in order to enable our targeted explorations are then to ensure our explorations can utilize open-source hardware, via implementing RISC-V support, and then developing extensible system models targeting AI solutions. In the next sections, I go over the effort to enable open RISC-V simulations with the gXR5 framework and then my extension to gem5-X for AI-targeted heterogeneous architecture explorations, the ALPINE framework.

2.3 gXR5: gem5 Extension for Linux-Capable Full System RISC-V Simulation

Owing to its highly modular and extensible nature, the RISC-V ISA is split into multiple different specifications, targeting different levels of system complexity and capability. These specifications contain instruction definitions organized by extension, hardware semantics, and other implementation details.

In order to run the most basic RISC-V system, e.g. a simple bare metal device, the Unprivileged Specification must be implemented, which includes extensions for integer, atomic, memory interface (load/store), etc., kinds of instructions [58]. At the time of gXR5’s creation, this
was the only specification implemented in gem5. But edge AI devices regularly employ full software stacks and operating systems, and therefore, cannot be run using only the RISC-V Unprivileged Specification. These systems need to additionally implement the RISC-V Privileged Specification, which includes new instructions for coherency and memory barriers, the concept of privilege modes, support for virtual memory, and specialized control and state registers (CSRs) [58]. Once the full Privileged Specification is implemented in the system model, running full system-level simulations requires a disk image, kernel binary, and bootloader. In the following subsections I detail the full system model implemented in gXR5 to enable simulation of Linux-capable systems. I then detail how this model was targeted and validated against a real RISC-V system, the SiFive HiFive Unleashed SoC. This work is detailed in [35] and later published in the RISC-V European Summit, 2023 [59].

2.3.1 gXR5 Full System Model and the SimpleBoard Platform

The RISC-V Full System model consists of numerous built-in gem5 models as well as newly-implemented ISA-specific models. As shown in 2.2, the model includes a single-core CPU with instruction and data caches, TLB page table walkers for each of the aforementioned caches, a (currently unused) boot memory, a Platform-Level Interrupt Controller (PLIC), a Core-Local Interrupter (CLINT), a PCI host with IDE controller, and lastly a UART module. All models are connected to either a memory bus (membus) or IO bus (iobus), and these buses are interfaced through both a system bridge and IO bridge. Of these models mentioned, only the PLIC and CLINT are RISC-V-specific models required by the ISA. Additionally, the PCI host is customized for our implementation. The rest of the models are ISA-independent and are provided by gem5’s built-in utilities.
Chapter 2  
Apparatuses and Tools: gXR5 and ALPINE

Figure 2.3: gXR5 model hierarchy.

The child tree is shown in 2.3 and shows the gem5 model hierarchy of the simulated system. This child tree controls the level at which different gem5 back-end interfaces can interact with each other, e.g., if the CLINT model wishes to interact with the snoop filter mode, it must be interfaced at the board level via the memory bus (membus).

Figure 2.4: gXR5 Full System-Level architectural model, showing the relationship between the CPU core, SimpleBoard SoC, and the memory subsystem.

To interface ISA-specific devices as well as external devices, I further develop the SimpleBoard platform that was initially introduced in Scheffel’s thesis [60]. The relation between the CPU
core and SimpleBoard SoC, as well as external devices, can be seen in 2.4. Broadly speaking, the SimpleBoard platform acts as the SoC for the simulated RISC-V board.

In addition to hosting SoC devices for RISC-V systems, the SimpleBoard platform is also used to interface PCI and console interrupts, as well as host a generic UART device. The PCI interrupt interface is required by our implementation for disk access; however, the console interrupt interface is currently unused. The SimpleBoard SoC is instantiated and connected to the main system through the gem5 FS mode configuration script.

### 2.3.2 RISC-V Unprivileged Spec Instructions, Routines, and Virtual Memory

#### Privileged and Unprivileged Specification Instructions in gXR5

gem5 implements instructions by using ISA template files that then generate all of the instruction classes and functionality during the SConstruct build process. The primary file used for implementing individual instruction functions is the RISC-V decoder.isa file, which decodes machine code to assign and process instruction types. Most of the RV64GC (RISC-V 64-bit machine general and compressed instruction set) instructions from the unprivileged ISA specification were implemented in prior work.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bit Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>URET</td>
<td>00000000</td>
</tr>
<tr>
<td>SRET</td>
<td>00000000</td>
</tr>
<tr>
<td>MRET</td>
<td>00000000</td>
</tr>
<tr>
<td>WFI</td>
<td>00000000</td>
</tr>
<tr>
<td>SFENCE.VMA</td>
<td>00000000</td>
</tr>
<tr>
<td>HFENCE.BVMA</td>
<td>00000000</td>
</tr>
<tr>
<td>HFENCE.GVMA</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Table 2.1: Instructions introduced in the RISC-V Privileged ISA specification and their bit layouts.

The privileged RISC-V spec adds very few instructions to the base ISA. These instructions are URET, SRET, MRET, SFENCE.VMA, WFI, HFENCE.BVMA, and HFENCE.GVMA. At the creation of gXR5, the Hypervisor specification had not yet been ratified, so we forego the implementation of the HFENCE instructions. The table of instruction layouts (Table 5.1 in the unprivileged specification document) is recreated in 2.1 for convenience [58].

The URET, SRET, MRET, modes. These instructions were initially implemented in prior work and only verified for use with Linux-capable FS mode in this work.

The WFI instruction is a "wait for interrupt" instruction. The RISC-V privileged specification states that a no-operation (NOP) is a valid implementation for WFI, so we left the instruction as such.

The SFENCE.VMA instruction is a supervisor fence that is used to enforce memory ordering.
While the RISC-V privileged specification goes into very tedious and specific detail about memory ordering and constraints, the result of the specification’s discussion is a relatively simple hardware cache flush, implemented in gem5 using the overridden demapPage method found in the RISC-V TLB. A call to SFENCE.VMA will flush a specific page or set of pages of the L1 ITB and DTB caches by address space number (ASN), virtual address, both, or neither (resulting in a full cache flush) depending on the values of its arguments rs1 and rs2.

```
fm  pred  succ  rs1  000  rd  0001111  FENCE
```

Table 2.2: Missing fence instruction from the RV32/64 Base Instruction Set.

```
imm[11:0]  rs1  001  rd  0001111  FENCE.I
```

Table 2.3: Missing fence instruction from the RV32/RV64 Zifencei Standard Extension.

Only two unprivileged instructions necessary for Linux-capable FS mode were missing from prior work. The instructions and their layouts are in Tables 2.2 and 2.3. The FENCE instruction is simply implemented as a flush instruction for the entire L1 instruction and data caches (ITB and DTB). The FENCE.I instruction is specifically for L1 ITBs, so only the ITB is flushed. The cache flush implementation in gem5 is the same as described previously for the SFENCE.VMA instruction.

### gXR5 Fault Handling and Algorithm

The RISC-V ISA specification defines numerous terms relating to trap/faults. A synchronous fault is referred to as an exception, while an asynchronous fault is referred to as an interrupt. Exceptions include faults due to misaligned instructions/addresses, invalid instruction/address access, illegal instruction calls, breakpoints, environment calls, and page faults. Interrupts include software, timer, and external asynchronous faults in different privilege modes. The fault number of an exception or interrupt is stored in the cause CSR, which holds fault causes in a one-hot representation and reserves its top bit to indicate if the fault is an interrupt or an exception.

Because RISC-V relies on its various binary interfaces for fault handling, a lot of the algorithm for handling various faults is delegated to software. This makes the implementation of fault handling in hardware extremely easy. The RISC-V hardware simply saves the current context (privilege mode, current PC value), escalates the privilege mode (unless delegated via the mideleg and medeleg CSRs), and sets the PC to the saved address of the fault handler. The cause value stored in the m/s/ucause CSR will tell the software fault handler which specific routine needs to be taken. When the fault is handled, it calls the m/s/uret instruction to restore context and continue program execution.

Prior work had defined and implemented RISC-V exceptions. All fault types derive from a base RiscvFault class and most faults will use the same invoke method which implements
the fault handling described in the previous section. The derived fault classes are mostly for implementation simplicity, but in this work we also had to ensure privilege modes were implemented for the ecall fault type. Address, misalignment, instruction, and page faults of all types are typically posted by the TLB, and are thus defined in the RISC-V TLB model.

Interrupts in RISC-V use the same fault handling algorithm as exceptions with some minor changes based on the fault code and cause CSR, and therefore use the same aforementioned fault handler defined for exceptions in gem5. To use the same fault handler, prior work had already defined and implemented an InterruptFault class. Interrupts are posted from different (usually external) sources at asynchronous times. gem5 offers a CPU interface with postInt and clearIInt methods that set and clear an interrupt pending array defined in src/arch/riscv/interrupts.hh.

Virtual Memory Subsystem

The satp (supervisor address translation and protection) CSR is responsible for determining the status of virtual memory. It contains three fields: the mode of translation, the address space identifier (ASID), and finally the physical page number of the root page table. When the mode is set to bare or the privilege mode is in M-mode, address translation is direct and all addresses are considered "physical". When the mode field is not empty, there are four modes of virtual address translation available.

In gXR5, we support only Sv39 virtual address translation. This is 39-bit virtual addressing that translates a 39-bit virtual address to a 56-bit physical address. The bit formats for Sv39 virtual address, page table entries, and physical addresses, are found in Figures 2.5, 2.6, and 2.7. Virtual address translation is mostly implemented in the RISC-V TLB model, which is described below.

Our gem5 TLB implementation is largely based on the TLB implementation for the ARM ISA, but with the RISC-V virtual address translation algorithm implemented. In other words, functional, atomic, and timings-based address translation calls are routed either through the
translateFs or translateSe methods. In our case, we move prior work to the translateSe method and focus on implementing the translateFs method, which is in charge of performing full system-level address translations in the context of a page table. The local page table of the TLB is stored in a CPP map data structure for simplicity. This data structure maps virtual address bases (virtual address without offset) to TlbEntry structures. The TlbEntry structures include the virtual address, physical address, page table entry, and ASID. The actual address translation takes place in the translateFs method of the TLB. This method is the main workhorse for all address translation types, PMP/PMA (Physical Memory Protection and Attributes) checking, and cache management with respect to other TLB methods.

When the satp CSR is set to bare translation mode, or the translation is occurring in machine mode, the physical address is translated directly and hence there is no need to cache the address. The address is simply checked against the PMP/PMA checker for potential access faults, before the physical address field of the requesting packet is simply set to the unchanged physical address.

When the satp CSR is not set to bare translation mode and the mode is supervisor or user, virtual address translation is required. The translation process starts with checking the TLB’s map data structure. If there is a TLB hit, we simply translate the virtual address using the cached physical address. If there is a TLB miss, we proceed with the virtual address translation algorithm explained in the RISC-V privileged ISA specification, section 4.3.2. The source code is annotated with each individual step of the virtual address translation algorithm.

Because a page table walker is a purely hardware component with no RISC-V specification, we use a very simple design that only acts to facilitate DMA transactions via the walk method. The table walker is set up using gem5’s ports interface to connect it both to the L1 caches and directly to the memory. Memory is accessed directly via the dmaAction method and the result is stored locally.

### 2.3.3 gXR5 Interruptor and SoC Models

#### Platform-Level Interrupt Controller

In our implementation, the PLIC is a BasicPioDevice on the SimpleBoard platform, and the first of our ISA devices. The configuration of the registers is defined as a map in the source code and follows the same layout at the PLIC described in the FU540-C000 core manual [61]. This layout is reproduced in the next section.

Additionally, all PLIC registers are 32-bit read-write registers, except for the pending array registers which are read-only. 64-bit registers are split into two 32-bit registers with a low and high field that is automatically interfaced by RISC-V software when accessing the PLIC. Note that this PLIC was designed for the HiFive Unleashed SoC with five harts (hardware threads; one small CPU core and 4 large CPU cores), even though our simulated FS mode system only
simulates one core (and therefore one hart) currently. We keep the registers for other harts for future work.

Each source number refers to a specific external device and designates a priority. For example, in the FU540-C000 manual, sources 1-3 refer to the L2 cache controller, source 4 refers to UART0, and so on. The values stored in the registers are 3-bit source priority values. A value of zero indicates that interrupts are disabled for the source, while a value of seven indicates that interrupts are of the highest priority for the source. Like with the additional registers for additional harts, we preserve all of the source registers for future work. In our FS mode simulation, we technically only use the sources for UART and the PCI host.

The PLIC interrupt pending array is a one-hot representation of pending interrupts where the bit index refers to the source number. For example, if bit 12 of the interrupt pending array is high, it means source 12 is awaiting interrupt handling.

The PLIC interrupt enable registers masks the available interrupts using the same one-hot representation used by the interrupt pending array. If an interrupt is pending and a hart is ready to handle an external interrupt, the hart will check the interrupt pending array against its assigned interrupt enable array, based on the current hart's number and privilege mode.

The PLIC priority threshold register holds a 3-bit priority value. In order for a hart of specific number and privilege mode to be able to handle an interrupt, its threshold value must be lower than the source priority value.

The PLIC claim and complete registers are used for the PLIC claims process, described in the next section.

At a high level, the PLIC is meant to perform two major operations in addition to standard reads and writes to its registers: PLIC claim, and PLIC claim complete. When a hart is ready to handle an external (global) interrupt, it ‘claims’ the interrupt (making it unavailable to other harts) by sending a read request to the hart's claim register. This will read the PLIC's interrupt pending array, clear the bit associated with the highest priority pending interrupt, and return the source number to the hart. The hart can then use the source number to jump to the correct interrupt handler for the source. When a hart is finished handling an external interrupt, it performs a ‘claim complete’, where it sends a write request to its claim/complete register. This register will hold the source number of the last completed external interrupt.

In our FS mode implementation, we use postInt and clearInt to send interrupts to the CPU. These methods are used primarily by the platform to forward PCI interrupts. Additionally, we only send M-mode and S-mode interrupts to the CPU; U-mode external interrupts can be delegated depending on the mideleg CSR. The sendInt method will send an external interrupt to the CPU if the source priority exceeds the source threshold as well as set the appropriate bit in the PLIC pending array. Likewise, the clearInt method will clear an interrupt in the CPU and clear the appropriate bit in the PLIC pending array.
Core-Local Interruptor

The CLINT, expanded from prior work [60], houses the system’s main clock. As a result, the CLINT is responsible for posting timer interrupts to its local CPU and associated harts. While there would usually only be one PLIC in a real system, there is usually one CLINT per hart. In addition to posting timer interrupts, the CLINT is also responsible for posting software interrupts. Our implementation loosely follows that of the CLINT implementation of the HiFive Unleashed SoC, so our CLINT will post only M-mode software interrupts [61]. As software interrupts are typically only used for cross-hart communication, they remain untested in gXR5.

Like the PLIC, our CLINT is implemented as a BasicPioDevice on the SimpleBoard platform that is also an ISA device. In gXR5, the CLINT is set up such that there is only one instance of the object in the whole system, but to handle multi-core and multi-hart systems in future work, we define a CpuTimer class that contains core-local and hart-local registers. The timer in the CLINT is, by default, set to 1MHz. The registers and layout of the CLINT are shown and described in the following section.

Like with the PLIC, the CLINT register layout follows that of the FU540-C000 core from the HiFive Unleashed SoC [61]. There is only one mtime register in the CLINT, but there is one mtimecmp and msip register per hart. All registers are read-write. Unlike the PLIC, the offsets of the mtimecmp and msip registers are generated as a function of the base offset added to the width of those register. For example, for hart 1 the offset of the interrupt pending (msip) and time compare (mtimecmp) CSRs would be 0x00000004 and 0x00004008, respectively.

The mtime CSR from the CPU’s register file is a memory-mapped CSR. A read/write operation to the CSR must perform the operation to the actual register location, which in our case, is the CLINT. The CLINT stores the mtime CSR as a simple 64-bit unsigned integer, which can be read/written to directly via address or via the ISA device interface. In a real (non-simulated) system, the CLINT’s timer would be tied to an external oscillator tuned to a constant frequency. Reading the mtime CSR means reading the timer register, which returns the number of cycles the oscillator has processed since system start. As gem5 is an event-based simulator however, implementing a timer directly in this way would be incredibly inefficient and increase simulation times significantly due to the number of added discrete events to the event queue. Therefore, the timer only changes with discrete events – primarily reads/writes to the CLINT. When the timer needs to be updated, it is done through the updateTime method. This method calculates the timer’s current value as a function of the default clock frequency of the timer and the current tick of the simulator. Note that it is also possible to write to the timer register, and therefore we preserve an offset value as well to incorporate into the timer update method.

The CLINT timer compare register, mtimecmp, is a register that determines when a timer interrupt must be posted. A timer interrupt is posted whenever the mtime CSR is greater than or equal to mtimecmp. Setting mtimecmp to INT MAX effectively disables timer interrupts. The mtimecmp register only determines when a timer interrupt is posted to the hart it is
attached to, and in a MPSoC, there would be one mtimecmp register per hart.

The msip register is supposed to hold one bit that indicates if a software interrupt is pending or not. In our implementation however, we tie the msip register directly to the mip CSR, and thus a read or write operation to the msip register in the CLINT is directly reflected in the mip CSR instead.

Timer interrupts occur whenever a CPU’s mtimecmp register is greater than or equal to the mtime CSR. In our implementation, we use gem5’s event queue to schedule "timer alarms" that go off and post either M-mode or S-mode timer interrupts to the CPU. Initially, the simulated system starts with mtime at 0 and timecmp at INT MAX, so timer interrupts are disabled. mtime will be modified any time it is read or written to by the simulation. When mtimecmp is written to, mtime is updated and an event (timer alarm) is scheduled for when mtime should be greater than or equal to mtimecmp. This timer alarm, when run, will verify mtimecmp and mtime, and then post a timer interrupt to the CPU. The software interrupt handler will then reset mtimecmp, which writes mtimecmp and consequently starts the timer alarm process again. If mtimecmp is written to again in between the aforementioned cycle, the previous timer alarm event is descheduled and a new one is scheduled in its place. Note that even setting mtimecmp back to INT MAX would still set a timer alarm, although at a tick extremely far into the future.

Miscellaneous SoC Models

For access to a disk image, gem5 uses a simulated PIIX4 IDE controller connected via PCI bus to a generic PCI host model. Our SimpleBoard SoC defines a GenericRiscvPciHost object that inherits from the base gem5 GenericPciHost object. The sole purpose of the custom PCI host is to map the correct source interrupt number from the PCI host to the PLIC. In our implementation, this is the base interrupt source number added to the PCI interrupt number. The base source number for the PCI host is 0x20, and thus the source numbers for all potential PCI interrupts are 0x21, 0x22, 0x23, and 0x24, although it is observed that only source number 0x21 is used.

The UART module is incorporated into the RISC-V system using the Uart8250 model that comes with the vanilla gem5 release.

2.3.4 gXR5 Validation

While full system-level simulations of Linux-capable RISC-V systems can reduce the time to market of novel RISC-V hardware and enable faster explorations of heterogeneous RISC-V systems for AI-enabled applications, these simulations only have merit if the performance statistics they produce are representative of real RISC-V hardware with a small margin of error. To this end, I contributed to the validation of gXR5 with respect to real hardware, specifically the first widely-available Linux-capable RISC-V SoC on the market SiFive HiFive.
Table 2.4: Simulator Model and HiFive Unleashed Technical Specifications

<table>
<thead>
<tr>
<th>Component</th>
<th>HiFive Unleashed</th>
<th>Simulated Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core</td>
<td>U54</td>
<td>MinorCPU</td>
</tr>
<tr>
<td>CPU ISA</td>
<td>RV64GC</td>
<td></td>
</tr>
<tr>
<td>CPU Frequency</td>
<td>1GHz</td>
<td></td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32kB</td>
<td></td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32kB</td>
<td></td>
</tr>
<tr>
<td>L2 Shared Cache</td>
<td>2MB</td>
<td></td>
</tr>
<tr>
<td>MMU</td>
<td>Sv39</td>
<td></td>
</tr>
<tr>
<td>Supported Modes</td>
<td>Machine, Supervisor, User</td>
<td></td>
</tr>
<tr>
<td>RAM Model</td>
<td>ECC DDR4</td>
<td>DDR4_4x16</td>
</tr>
<tr>
<td>RAM Frequency</td>
<td>2400MHz</td>
<td></td>
</tr>
<tr>
<td>RAM Size</td>
<td>8GB</td>
<td></td>
</tr>
<tr>
<td>System Bus</td>
<td>TileLink</td>
<td>XBar</td>
</tr>
<tr>
<td>Storage Access</td>
<td>QSPI</td>
<td>PCI</td>
</tr>
<tr>
<td>Interrupters</td>
<td>CLINT and PLIC</td>
<td></td>
</tr>
</tbody>
</table>

Unleashed SoC (HF1), using select benchmarks from the stress-ng microbenchmark and the SPEC CPU2017 benchmark suites [62], [63]. This validation effort was ultimately published in the 2023 RISC-V Summit [59].

Validation Experimental Setup

The HF1 board is a quin-core system with four large U54 in-order cores and one small U51 in-order core. The large cores are typically responsible for user-space applications while the small core handles interrupts and OS services. For this validation effort we target specifically a single U54 core.

Table 2.4 shows a comparison between the simulated model in gXR5 and the real hardware. While we can simulate most components similarly, there are fundamental components – mostly in the memory and IO subsystem – that cannot be targeted for validation. Therefore, for this validation effort, we need benchmarks that specifically target the U54.

stress-ng Microbenchmark and SPEC CPU 2017 Benchmark Selection

Because of gXR5’s modularity, virtually every component can be tuned to be validated against real hardware. Functional unit operation latencies, pipeline stage bit-widths, the number of functional units, branch predictor size and algorithm, and so on, are all configured in gXR5’s configuration scripts. Unfortunately, the manual for the HF1, which we target for validation, does not give us all of the details we need to be able to tune every gXR5 configuration parameter, and thus perform a complete validation of the full system. For example, we know that the HF1 contains 32kB of L1 data and instruction caches, but we do not know how many multiplier
units are housed within the U54. So given potentially hundreds of configuration parameters to tune in simulation for hundreds of black boxes, we must narrow down the scope of the validation.

As a first step in the validation effort, we comb over the HF1 manual to tune as many known parameters as possible – this includes branch predictor size, the number branch target table entries, CPU core frequency, cache sizes and associativities, CPU functional unit operation latencies (for adder, multipliers, dividers, and load/store units), and CLINT timer frequency. Then to start testing these parameters, we use the stress-ng microbenchmark suite [62].

Stress-ng is a software-level microbenchmark suite typically used for hammering edge conditions in a variety of hardware and software scenarios. It includes hundreds of microbenchmarks testing things from virtual memory page swapping, to OS interfaces, to branch predictors and other hardware components. While the ultimate goal is to validate the gXR5 CPU model on real benchmarks (i.e., the SPEC CPU 2017 benchmark suite), normal benchmarks take too much simulation time in order to quickly iterate over parameter variations. With stress-ng, we can simultaneously limit the run time of microbenchmarks (via limiting iterations, or in stress-ng terms, "Bogo-ops") as well as limit the number of parameters needing tuning per microbenchmark.

However, stress-ng comes with hundreds of microbenchmarks targeting different portions of the system. Even if we utilize only CPU microbenchmarks, there are still over 100 microbenchmarks available. To further limit the number of benchmarks we need to validate gXR5, I run all of the CPU microbenchmarks once in gXR5 to gain initial statistics, including instructions per cycle, cache hit and miss rates, number of instructions run, and mix of instructions run.

After selecting the most relevant statistics, I vectorize each stressor’s statistics and place said vector into a random forest classifier to find 10 microbenchmarks that best represent the whole set (of over 100 stressors and methods). For every stressor in a subset of 10, they are ranked according to their similarities with the rest of the stressors. The stressors with the best rank over all iterations of all subsets of stressors are then selected for component-level calibration to help tune particular aspects of the RISC-V system.

For the full validation, then, we target the SPEC INT suite from the SPEC CPU benchmarks. Due to constraints in the file system and library dependencies, we only run five of the ten SPEC CPU 2017 benchmarks with test inputs. The resultant benchmarks and microbenchmarks are listed in Table 2.5.
**Stress-ng Microbenchmarks**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prime</td>
<td>Find the first 100,000 prime numbers using a brute force trial division search.</td>
</tr>
<tr>
<td>sqrt</td>
<td>Find the square root of double and long double type numbers less than 16,384.</td>
</tr>
<tr>
<td>queens</td>
<td>Solve the &quot;Queens&quot; problem for sizes 1 to 12.</td>
</tr>
<tr>
<td>rand48</td>
<td>16384 iterations of stdlib's drand48 and lrand48 methods.</td>
</tr>
<tr>
<td>matrixprod</td>
<td>Matrix product of two 128^2 matrices of double float type.</td>
</tr>
<tr>
<td>longdouble</td>
<td>1000 iterations of mixed-precision floating point operations</td>
</tr>
<tr>
<td>stats</td>
<td>Calculate the minimum, maximum, various means, and standard deviation on 250 randomly generated positive double type values.</td>
</tr>
<tr>
<td>trig</td>
<td>Compute $\sin(x)\cos(x)+\sin(2x)+\cos(3x)$ for float, double, and long double-type functions where x is 1500 steps in the range $[0, 2\pi]$.</td>
</tr>
<tr>
<td>intconversion</td>
<td>Perform 65536 iterations of integer conversions between int16, int32, and int64 variables.</td>
</tr>
<tr>
<td>int64longdouble</td>
<td>1000 iterations of a mix of int64 and long double type floating point operations.</td>
</tr>
</tbody>
</table>

**SPEC CPU2017 Benchmarks**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>505.mcf_r</td>
<td>Based on MCF, a program used for public transport scheduling.</td>
</tr>
<tr>
<td>520.omnetpp_r</td>
<td>Discrete event simulation of a large ethernet network.</td>
</tr>
<tr>
<td>523.xalancbmk_r</td>
<td>An XSLT processor written in C++.</td>
</tr>
<tr>
<td>531.deepsjeng_r</td>
<td>AI alpha-beta tree search applied to Chess.</td>
</tr>
<tr>
<td>541.leela_r</td>
<td>AI Monte Carlo tree search applied to Go.</td>
</tr>
</tbody>
</table>

Table 2.5: Selected Benchmarks for gXR5 Validation ([62], [63])

**Validation Methodology**

Figure 2.8: Intensity-level classifications of microbenchmarks utilized to guide gXR5 validation.
The benchmark intensity classification for gXR5 is shown in Figure 2.8. Stress-ng microbenchmarks are first classified as control, memory, or arithmetic intensive for CPU parameters by profiling both the static binary and ROI statistics generated by an un-calibrated gXR5.

Control intensive microbenchmarks have complex control structures (e.g. nested for-if statements) with a minimum 100 branches per 1000 instructions. Memory intensive microbenchmarks have at minimum 20% load/store instructions in the op-code mix, with a maximum of 30% for "queens" and "rand48". Lastly, arithmetic intensive microbenchmarks have a minimum 20% integer/float instructions in the op-code mix.

The general methodology is as follows:

- We first select a particular intensity class to target for validation.

- Next, we identify the models most relevant to the component class using ROI statistics generated by gem5 as well as static analysis of the stressors that are within that component class. For example, in the control-intensive component class, we identify numerous control structures which are classically most affected by branch predictor specifications and algorithms, and therefore we go over the parameters of the branch predictor model.

- After identifying the specific model we wish to tune for validation, we perform iterations on the stressors. This involves changing model-specific parameters, recompiling gem5, and then launching parallel simulation of the stressors – a process that only takes a couple of hours (as opposed to the days or weeks it may take to run a full benchmark suite). The stressors are run for five simulated seconds each in gXR5.

- When an iteration is finished, the IPCs and run times are compared to measure the new error given the parameters that changed. If the error increased, the changed parameter is reverted back to its original value before the iteration. If the error decreased, the changed parameter is kept for future iterations.

- When the error has been brought down significantly over the baseline (e.g., by 5-10% with a small set of tuned parameters), the next component class is targeted for validation. This may involve returning to prior component classes if new bottlenecks are identified during the validation process.

- Finally, once validation with the microbenchmarks is complete, the results are checked using full SPEC CPU2017 benchmarks.

The general progression towards our validated gXR5 can be seen in Figure 2.9. The summation of all the tuned parameters can then be seen in the next subsection, in Table 2.6.
We started the validation effort with the arithmetic-intensive stressors. As these stressors largely use arithmetic integer instructions, we targeted parameters residing in the functional units of theMinorCPU model, which include op latencies and number of functional units. In this step, control- and arithmetic-intensive stressors achieved 5% less IPC error over the un-calibrated (baseline) gXR5.

Once no more decrease in IPC error could be achieved, we moved to validate the arithmetic-intensive stressors by tuning parameters in the branch predictor model, including the size of the branch table buffer, branch history table, return address stack, and others. This further decreased the IPC error of control- and arithmetic-intensive benchmarks by another roughly 5% as compared to baseline.

Seeing that the memory-intensive stressors saw barely any change in IPC with the previous two sets of iterations, we then moved to validate the cache models by changing cache associativity and latency parameters. This brought down the IPC error of the memory-intensive stressors by 22%. The further control- and arithmetic-intensive benchmarks also saw another roughly 5% IPC error decrease.

Finally, having addressed the most direct and obvious models for validation, we performed one final sweep of MinorCPU model pipeline parameters, which would affect all of the stressors. These parameters included line widths in between stages, delay cycles, and the number of entries in various buffers. The arithmetic-, control-, and memory-intensive stressors saw an average IPC error decrease of 2%, 3%, and 10%, respectively.

Figure 2.9: gXR5 validation at the end of each set of iterations dedicated to validating functional units, branch predictor, memory, and pipeline models.
Final Validation Results

The pre-validation and final post-validation parameters for the gXR5 model are shown in Table 2.6.

<table>
<thead>
<tr>
<th>gXR5 Parameter</th>
<th>Pre-Validation Value</th>
<th>Post-Validation Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU Stages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Fetch1 Line Width</td>
<td>64B</td>
<td>16B</td>
</tr>
<tr>
<td>CPU Fetch2-Fetch1 Backward Delay</td>
<td>1 Cycle</td>
<td>0 Cycles</td>
</tr>
<tr>
<td>CPU Decode Input Width</td>
<td>2B</td>
<td>4B</td>
</tr>
<tr>
<td>CPU Decide Input Buffer Size</td>
<td>3 Entries</td>
<td>4 Entries</td>
</tr>
<tr>
<td>CPU EX Input Width</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CPU EX Issue Limit</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CPU EX Commit Limit</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CPU EX LSQ Transfer Queue Size</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>CPU EX Memory Width</td>
<td>64B</td>
<td>16B</td>
</tr>
<tr>
<td>CPU EX Max Access Memory</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td><strong>RISC-V Instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Op Latency</td>
<td>4 Cycles</td>
<td>2 Cycles</td>
</tr>
<tr>
<td>LH/LB-type Op Latency</td>
<td>4 Cycles</td>
<td>3 Cycles</td>
</tr>
<tr>
<td>MUL-type Op Latency</td>
<td>1 Cycle</td>
<td>1 Cycle</td>
</tr>
<tr>
<td>DIV/REM-type Op Latency</td>
<td>33 Cycles</td>
<td>19 Cycles</td>
</tr>
<tr>
<td><strong>Branch Predictor</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP Branch Table Buffer</td>
<td>4096 Entries</td>
<td>30 Entries</td>
</tr>
<tr>
<td>BP Branch History Table</td>
<td>2048 Entries</td>
<td>256 Entries</td>
</tr>
<tr>
<td>BP Return Address Stack</td>
<td>16 Entries</td>
<td>6 Entries</td>
</tr>
<tr>
<td>BP Hit Latency</td>
<td>3 Cycles</td>
<td>1 Cycle</td>
</tr>
<tr>
<td>BP Miss Penalty</td>
<td>3 Cycles</td>
<td>3 Cycles</td>
</tr>
<tr>
<td>BP Algorithm</td>
<td>Tournament</td>
<td>Multiperspective Perceptron</td>
</tr>
<tr>
<td><strong>Caches</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1D/I Cache Associativity</td>
<td>2-way</td>
<td>8-way</td>
</tr>
<tr>
<td>L1D Cache Data Latency</td>
<td>3 Cycles</td>
<td>1 Cycle</td>
</tr>
<tr>
<td>L1D Cache Response Latency</td>
<td>5 Cycles</td>
<td>2 Cycles</td>
</tr>
<tr>
<td>L1D Clusivity</td>
<td>Mostly Exclusive</td>
<td>Mostly Inclusive</td>
</tr>
<tr>
<td>L2 Cache Associativity</td>
<td>8-way</td>
<td>16-way</td>
</tr>
</tbody>
</table>

Table 2.6: gXR5 Validation Parameter Values

The results of the calibration and validation efforts for the stress-ng microbenchmarks and SPEC CPU 2017 benchmarks are shown in Figures 2.10 and 2.11, respectively. Using the stress-ng microbenchmarks, we are able to bring down the average IPC error from 36% to 11.8% in the gXR5 model with respect to real hardware. As a result, the execution time error of the SPEC CPU2017 benchmarks is reduced from 44.3% to 23.9% [59]. It is worth emphasizing that the SPEC CPU2017 benchmarks take on the order of 3-7 days to finish one validation iteration.
Figure 2.10: IPC of the HF1 board, pre-validation gXR5 model, and post-validation gXR5 model for each of the stress-ng benchmarks.

Figure 2.11: Execution time of the HF1 board, pre-validation gXR5 model, and post-validation gXR5 model for each of the SPEC CPU2017 benchmarks. The #XX indicates the percent of load/store instructions within each benchmark.

(change parameters, recompile gem5, run benchmarks), as opposed to the 1-2 hours to finish one iteration of the stress-ng stressors.

2.4 The ALPINE Framework: Expanding gem5-X and RISC models for exploring Accelerators and Interfaces

A fundamental trade-off that needs to be considered when approaching AI-enabled applications is the trade-off between flexibility, capability, and proximity. Due to their large memory footprint, AI applications suffer greatly from data movement-based bottlenecks. Therefore, heterogeneous edge AI architectures typically try to reduce the data movement or move compute closer to the data [48], [64]. Edge AI architectures still predominantly use the CPU for compute resources and orchestration [65], and so we can model this trade-off at the hardware
level by adding a new generic peripheral input-output (PIO) device to gem5-X, the basis for ALPINE.

### 2.4.1 Generic Peripheral Input-Output Device Implementation

![Diagram of ALPINE Wrapper Object]

Figure 2.12: ALPINE Wrapper Object. Input control and output can refer to multiple kinds of interfaces, including DMA, IO bus, and ISA extension, and are meant to orchestrate accelerator functions.

The backbone of ALPINE is the generic PIO device that can be extended to model a variety of edge AI solutions, which is shown in Figure 2.12. This PIO device is an object that sits atop the RealView simulated SoC for ARM systems and connects with the rest of the system via the IO bus.

All PIO devices must implement a read and write method for interacting with packets sent via the bus. These methods return the delay in response from the device to the packet sender, which can be used to model variable delays in accelerator operation. The PIO wrapper is assigned to an address range so that reads and writes to specific addresses can be parsed for specific operations, including accelerator configuration, or in the case of flexible experiments in Chapter 4, dynamically resizing the accelerator during run time to very quickly reassign resources.

However, the IO bus-based interface presents a bottleneck to data movement: its predetermined bit-width, response latency, frequency of operation, and all other devices attached to it all contend with this resource. Novel accelerators are starting to have much closer integration to the CPU, such as in [29]. Therefore, we connect the PIO device to the CPU via the system object. In this configuration, we connect the system object to the PIO device itself, enabling set and get methods on the system object that are able to call methods from PIO device.

Theoretically, any object attached to the system itself can then access the ALPINE PIO device directly with a separate timing model. Of interest to us though is the ability to interface the PIO device directly from the CPU, e.g., from an ISA extension. To this end, we utilize gem5-X’s ISA extension to interface the PIO device. Each of the custom instructions in the gem5 domain-specific language has a link to the system object in ALPINE, which is then used to query the PIO device directly. Timings can be modeled in terms of CPU cycles by adjusting the
Chapter 2  Apparatuses and Tools: gXR5 and ALPINE

CPU’s functional unit operation latency.

With this framework in place, it can then be specialized towards specific AI solutions. I go over the modeling of specific AI solutions using the ALPINE framework in Chapters 3 and 4 for AIMC tiles and systolic arrays.

2.4.2 Software Support: AIMClib and tinytensorlib

To assist in the integration of the PIO device’s interfaces into user-space applications, we develop two software libraries to quickly integrate accelerator functions in neural networks.

The first of these is AIMClib. Although it is built specifically for interfacing analog in-memory compute (AIMC) tile models (presented in Chapter 3), fundamentally these instructions handle multiple types of ARMv8 ISA extension-based transfers to and from the ALPINE PIO device. AIMClib is a C++-based library that puts the ISA extension intrinsics into wrappers for easier use. It then includes methods to facilitate large data transfers to and from arrays of various types. In [39], it is even extended to facilitate data transfers to and from data structure of the highly-optimized Eigen C++ library [66]. Each of these intrinsics can be accessed individually which facilitates refactoring of the accelerator model.

To augment AIMClib, we then introduce tinytensorlib. tinytensorlib is a minimalist neural network library used for quickly spinning up neural network models for inference that use AIMClib’s ISA extension. It also features a compilation flag to automatically switch between the ISA extension-based interface initially implemented with AIMClib and the more traditional bus-based interface with memory-mapped IO (MMIO). This is visualized in Figure 2.13.

To ensure the neural networks built with tinytensorlib are on-par, performance-wise, with modern neural network frameworks such as PyTorch or tensorflow, tinytensorlib utilize Eigen C++ library tensors as the back-end for all inputs, outputs, and weights and operations. It also supports ping-pong buffering in case of multi-threaded NN applications.

In summary, tinytensorlib can be used to quickly spin up accelerator-enabled and digital
Apparatuses and Tools: gXR5 and ALPINE

Chapter 2

reference neural networks for inference performance testing within the ALPINE framework, with support and examples for multi-layer perceptrons, classic convolutional neural networks, resnets, mobilenets, and more.

2.4.3 Power Modeling

For full system power models, we are able to leverage the same one used for gem5-X [28], derived from McPAT [67]. In general, the energy and power calculations for ALPINE-based simulations are a combination of CPU active power, CPU wait-for-memory power, CPU idle power, last-level cache power, and memory power. These statistics are given in per cycle or per byte metrics, which can be combined with the output statistics file of ALPINE simulations to get total energy and power.

2.5 Summary

In this chapter, I presented the gXR5 and ALPINE frameworks to enable architectural explorations of heterogeneous edge AI systems. gXR5 enables the world’s first Linux-capable full system-level simulation of RISC-V systems by implementing the RISC-V privileged ISA specification and associated hardware models. gXR5 was validated against real RISC-V hardware, the SiFive HiFive Unleashed SoC using a combination of microbenchmarks and benchmarks from the stress-ng and SPEC CPU2017 suites to lower the run time validation error from 36% to 11.8% and 44.3% to 23.9%, respectively. I then went on to implement the ALPINE framework, which integrates a new generic peripheral input-output (PIO) device model that includes multiple interfaces for modeling the proximity and flexibility of modern AI solutions and accelerators. Using gem5-X’s ISA extension, I am able to connect CPUs directly the accelerator and model close connections, while simultaneously offering a bus-based interface for DMA and memory-mapped IO connections. To facilitate the process of building highly optimized software for Neural Network inference performance testing, I built two libraries, AIMClilib and tinytensorlib. AIMClilib initially provided library methods to easily integrate the ISA extension into modern neural network applications. tinytensorlib expanded on AIMClilib to add support for highly optimized neural network implementations and compilation options to build applications for different kinds of interfaces. These contributions are ultimately what led me to our first set of architectural explorations of tightly-coupled (ISA-based) interfaces in Chapter 3.
3 Tightly-Coupled AI Accelerator Solutions

3.1 Introduction

Deep neural networks (DNNs) have revolutionized the state-of-the-art in a wide range of AI applications ranging from computer vision to natural language and speech processing. DNNs are composed of multiple consecutive layers, and their ability to address tasks often increases with the number and size of layers. Today, modern DNNs are composed of hundreds of layers and millions of weights, requiring massive amounts of computation and memory [68], [69]. In the quest for achieving higher accuracy and throughput, the AI domain suffers from constant changes in the type and nature of the DNNs.

One of the main contributing factors to the expansion of DNNs has been the introduction of more powerful CPUs and GPUs. However, system solutions based on CPU-GPU integration struggle to meet the efficiency requirements of the edge domain, where the extreme data-intensive nature of DNNs mandates efficient storing, accessing and processing of large amounts of data. Efficiency gains can be achieved by lowering data precision (e.g., from 64/32-bit floating point to 8/4-bit integer) [70]. Yet, the time and energy for accessing the data still dominate over the data processing [71]. Hence, solutions for ultra-efficient DNN processing require a re-thinking of the architecture of computing systems, as well as the adoption of new paradigms at the hardware level.

In particular, recent technological breakthroughs in the field analog in-memory computing (AIMC) as well as systolic array-based accelerators blur the distinction between processing and memory with custom-designed memory, which look beyond conventional von Neumann architectures. With AIMC, certain computations directly take place where the data is located, exploiting device physics and circuit laws [72]. In addition to significantly reducing the data movement, AIMC core enables the execution of millions of operations (such as multiply-and-accumulate, MACs) in parallel, greatly outperforming GPUs and other accelerators. In contrast, Systolic Array (SA) architectures [73] are the focus of renewed research interest [32] because they target parallel execution of general matrix-matrix multiplication operations (GEMMs) on a 2-dimensional mesh of processing elements, computing outputs in linear time.
Chapter 3  Tightly-Coupled AI Accelerator Solutions

One approach to exploit these accelerators for edge DNNs is to design stand-alone accelerators where multiple AIMC tiles or systolic array blocks are interconnected by a suitable communication fabric with additional specialized digital hardware logic or IO buffers [74]–[76]. However, under typical loosely-coupled interfaces, e.g. kernel driver, IO bus, PCI bus, DMA controller, and so on, any communication overhead with the CPU translates into under-utilized AIMC tile or systolic array compute cycles and hence a performance loss. Therefore, it is essential to work towards a tight integration to fully realize the potential of these acceleration methods, including support for a software ecosystem so that systems can be easily configured to implement a wide range of neural networks and be compatible with the commonly used ML frameworks.

Towards this end, I use ALPINE to explore multiple tightly-coupled CPU-accelerator architectures and extrapolate their trends in performance and energy, as well as compare them directly to each other in common NN applications. My contributions in this chapter are as follows:

- I use the ALPINE framework and AIMClib to model tightly integrated AIMC tiles, governed by custom instructions extending the ARMv8 64-bit instruction set architecture (ISA) with custom instructions.

- I showcase the mapping of different artificial neural network (NN) types (MLP, LSTM, CNN) onto the proposed AIMC tile-enabled architecture in both single-core and multi-core cases. I obtain up to 20.5x performance and up to 20.8x energy benefits (in CNNs) with respect to the multi-threaded CPU + SIMD (ARM NEON) implementation.

- Using the aforementioned NNs as case studies, I analyze the prevalence of matrix-vector multiplication (MVM) operations as a computational hot-spot in DL workloads, and quantify the application-wide benefits achievable by acceleration via tightly-coupled AIMC tiles, including up to 12.8x/12.5x speedup and energy improvement in MLPs and 9.4x/9.3x speedup and energy improvement in LSTMs.

- To show the flexibility of the ALPINE framework and its ability to model numerous edge AI solutions, I contribute to a study of tightly-coupled systolic arrays (TiC-SATs) by providing the custom PIO device model with ISA connections. By developing a new data-flow model for systolic arrays using a refactored AIMClib, primary author Alireza Amirshahi was able to show up to 89.5x speedup gains for cutting-edge Transformer NN applications.

- Given the two aforementioned edge AI accelerator models implemented with ALPINE, I am able to perform the first direct experimental analysis of both technologies for a common application. Using the Chatfield CNNs as the target NN, I compare the usability, implementation, nuances, performance, and energy benefits of using AIMC tiles and TiC-SATs over SIMD-enabled edge systems or systems that simply use larger caches. My analysis highlights the trade-offs in flexibility, capability, and speedup that
come with specific performance gains from AIMC, systolic array, and SIMD systems, where AIMC tiles gain a maximum 9.7x speedup over large-cache systems.

3.2 Related Work

3.2.1 Analog In-Memory Compute Tiles

Research efforts investigating AIMC for inference have, for the most part, focused on achieving classification accuracy on-par with digital systems (iso-accuracy). Prior work includes iso-accuracy studies for convolutional neural networks (CNNS) [77]–[79], recurrent neural networks (RNNs) [78], [80] and transformers [81]. However, very few papers have discussed how to model the potential benefits of AIMC at the system level or discuss the integration of AIMC for system acceleration.

Most of the works in this space are illustrated as part of architectural and compilation frameworks studies, and are therefore tightly linked to a hardware/software ecosystem. Among them, Chen et al. introduce an instruction-accurate-only circuit-level simulator for gathering statistics on area, latency, dynamic energy, and leakage power of a synaptic array model [82]. Working at a higher level, Kourtis et al. [83] introduce a software stack to automate the mapping of CNNS (described in high-level language) onto multi-tile AIMC accelerators. As part of their contribution, the authors showcase a cycle-accurate simulator. Similarly, Ankit et al. [84] describes an ISA and compiler dedicated to programming and utilizing a multi-tile AIMC accelerator, and an associated architectural simulator (named PUMASim) to evaluate the energy and performance of compiled applications. The scope of both these approaches is limited to the modelling of the AIMC accelerator alone, neglecting other system components.

Ambrosi et al. [85] propose, as part of their ONNX compilation framework, a set of simulators at three abstraction levels: performance (low-level hardware simulation of single AIMC instructions), functional (component-level simulation of AIMC tiles and associated memories) and system. Our research contribution is most closely related to the latter, which is based on the QEMU emulator and aims at investigating the execution performance of entire applications. Zheng et al. also use the ONNX framework as the front end for their event-driven cross-level simulation of processing-in-memory accelerators, while also incorporating elements for simulating memory access and interconnects [86]. However, both of these simulators do not take into account the interaction between applications and operating systems, nor does it consider the interplay between AIMC tiles and the rest of the system, including CPUs and, in the case of [85], the memory hierarchy.

Most closely related to my approach is the paper of Vieira et al., which details a full-system evaluation strategy of AIMC acceleration. As in this case, the authors also base their approach on AIMC-dedicated extensions to the gem5 environment [29]. Nonetheless, their approach is limited to modelling the simple case of binary CNNs, and their per-kernel mapping strategy does not scale to the larger and more general applications I tackle in this paper. Moreover, as
opposed to our ISA extension enabling multi-core CPU systems with multiple AIMC tiles, their extension only supports a single-core CPU and a single AIMC tile.

Finally, Ottavi et al. [49] proposes and synthesizes a heterogeneous architecture where an AIMC tile sits within a cluster of RISC-V processors. They perform a design-space exploration using the bottleneck of MobileNetV2 CNN where point-wise and depth-wise convolutions are placed within the AIMC tile. However, their synthesis approach hinders quick exploration of alternative architectures and model runs.

### 3.2.2 Systolic Arrays for Edge AI Inference

Systolic arrays are recently gaining popularity as a solution to edge AI inference, with several works being published within the last several years. Most of these works propose a "loosely-coupled" model far from the CPU and interfaced via buses or similar that may or may not explicitly target the edge domain. For systolic arrays, the main drawback in using loosely-coupled interfaces is the very high hardware memory requirement in order to leverage the accelerator functions.

One of the earlier works is [76], which proposes a systolic array model for cycle-accurate simulation and deep learning. More recently, Gemmini [32] and SMAUG [30] are platforms that utilize systolic array-based accelerators for full system-level simulation. In these platforms, the area requirements for memory are massive: scratchpad software-programmable memories for use with the accelerators represent 52.9% and 79.1% of the total area for the accelerators for Gemmini and SMAUG, respectively.

[87] tries to improve upon this with an algorithm designed to dynamically identify weights most critical for performance in Transformers followed by hardware weight filtering. Following aforementioned trends, 58.6% of the area is required to be buffers for input and output data.

To the author's knowledge, the work on TiC-SAT is the first known attempt to tightly couple small systolic arrays with CPUs for better performance and lower area overhead.

### 3.3 Analog In-Memory Compute Tiles for Machine Learning Inference Workloads

#### 3.3.1 Analog In-Memory Compute Tiles

**Analog In-Memory Computing Paradigm**

AIMC offers significant advantages in terms of energy and performance owing to two key properties. First, the computation takes place in the memory and therefore, the expensive data movement can be avoided (addressing the memory read bottleneck). Secondly, the computation can be done in a massively parallel and analog manner by exploiting the physical
attributes and state dynamics of memory devices. SRAM-based AIMC approaches are attractive owing to SRAM’s technological maturity and scalability to aggressive CMOS nodes [88], [89]. One drawback with this approach is that only a single bit can be stored in an SRAM cell. An alternative is to adopt AIMC based on non-volatile memory technologies, including 2D [90] and 3D Flash [91], phase-change memory (PCM) [92], and resistive random-access memory (RRAM) [93]. These technologies offer analog data storage capability, i.e., the ability to achieve a continuum of resistance/conductance values (Fig. 3.1(a)). Their non-volatile nature makes them particularly attractive for low-power embedded applications as non-volatile memory-based AIMC tiles consume negligible static power.

In this thesis, I focus on PCM for AIMC, which is arguably the most mature technology among the class of resistance-based or memristive memory devices (Fig. 3.1(a)). PCM devices have the potential to scale to nanoscale dimensions and can be integrated in the back-end of a CMOS chip [92]. Therefore, PCM-based implementations offer high performance densities (TOP/s/mm²), where a pair of PCM devices can represent signed multi-bit weights [77].

MVM operations, which form the bulk of computation for DNN models, can be implemented in a PCM crossbar by representing the elements of a $M\times N$ matrix as the conductance values of memory devices, as shown in Fig. 3.1(b). Each element of an input vector is translated into the duration of a voltage pulse with fixed amplitude $V$. The voltage pulses are applied simultaneously to $M$ word lines and each memory device contributes to the current flowing through one of the $N$ bit lines, with an amount directly proportional to its conductance $G$ (Ohm’s law). The total current integrated on each of the bit lines over a certain period of time $t_{int}$ is indicative of the result of the dot product between the $M$-element vector and a column of the $M\times N$ matrix (Kirchoff’s current law). Hence, the multiplication of an $M\times N$ matrix with an $N$-element vector can be performed in a constant amount of time, (in the range of 10s to 100s of nanoseconds [92]).
Analog In-Memory Compute Tile Architecture

An AIMC tile contains digital-to-analog and analog-to-digital converters (DACs and ADCs) with dedicated registers and a local controller, alongside the memory crossbar array of unit cells, as presented in [92]. DACs convert the signed digital input into a voltage pulse; the pulse amplitude is applied either as positive or negative differential input ($+V$ or $-V$) according to the sign and the duration of the pulse is proportional to input magnitude. The dot product over the bit line is converted to a signed digital output via ADCs. Each signed weight is represented with a pair of PCM devices; therefore, a differential bit line current is received by the ADC. The local controller orchestrates the data flow from the data bus into the DAC registers and out of the ADC registers to the data bus. It also activates the MVM operation when input data has arrived into the DAC registers.

In this design, there is a dedicated DAC and ADC for each word line and bit line, respectively. The resolution of DACs and ADCs are signed 8-bits. The input signal is scaled and quantized in digital prior to its transfer to the AIMC tile. This input scaling factor can be arbitrarily selected, preferably fixed to avoid dynamic scaling. Similarly, the ADC quantizes the output of the MVM operation.

Computation Precision with Analog In-Memory Computing

In addition to the quantization introduced by DACs/ADCs, the weights stored in the memory crossbar also have a low precision. Yet, the nature of the precision loss for the weights with analog computing is substantially different from the weight quantization of a digital implementation. The programming and reading of analog weight value are prone to various non-idealities, including noise, temporal conductance fluctuations and temperature-induced variations [94], [95].

The scalar multiplication of an analog input with PCM-based weights is shown to be comparable to an implementation with 4-bit fixed-point inputs and weights [96], and even to an implementation with 8-bit fixed-point inputs and weights with suitable innovations in device design [97]. To counter the reduced weight precision, noise can be added during training, so that the model is more robust when performing inference on AIMC tiles [77]. An alternative approach is to encode weights using multiple PCM devices [80]. Despite the reduced precision weights, AIMC implementations were shown to address the inference of MLPs [94], [95], CNNs [77], [94], [95], RNNs [80], [94], [95], and transformers [81] with high accuracies.

3.3.2 AIMC Tile-Enabled System Architecture

AIMC Tightly- and Loosely-Coupled Integration

A high-level overview of a loosely-coupled AIMC-enabled system, can be seen in [31], [98]. The AIMC tiles are peripheral devices that communicate with the CPU cores via the I/O bus.
cores use load and store instructions to read and write data to a particular memory-mapped addresses. The AIMC tiles are typically organized as a multi-tile accelerator with a control unit that parses the incoming data so that one or multiple AIMC tiles within the accelerator can be accessed.

While a more common system design approach, loosely-coupling AIMC tiles in this fashion typically leads to a significant communications overhead that can cause the CPU to stall and wait for transactions (empirical data for this assertion is presented in 3.3.5, and then elaborated upon in Chapter 4). This is typically overcome by placing more digital logic on the accelerator containing AIMC tiles, but this strategy comes with numerous drawbacks, including more accelerator energy and power requirements, more fabricated hardware area for the digital logic, and hard constraints on the digital logic capability (i.e., a dedicated ReLU functional unit included in the accelerator model will not be able to handle Sigmoid activation functions or more complex operations that could be required by the network).

To overcome the communication overhead over the bus, as well as the constraint of flexibility in the loosely-coupled design, I propose a novel tightly-coupled configuration, as seen in Fig. 3.2. Here, the CPU uses an ISA extension to access private AIMC tiles that are unique to each of the CPU cores, without requiring the traversal of the memory hierarchy (Fig. 3.2).
(a) High-level Data-flow with ISA Extension

![Data-flow visualization](image)

(b) Proposed CM Instruction Definitions

<table>
<thead>
<tr>
<th>Op</th>
<th>OpCode</th>
<th>Rm</th>
<th>R/W</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM_QUEUE</td>
<td>0x108</td>
<td>Rm</td>
<td>1</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>CM_DEQUEUE</td>
<td>0x108</td>
<td>Rm</td>
<td>0</td>
<td>X</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>CM_PROCESS</td>
<td>0x008</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Rd</td>
</tr>
<tr>
<td>CM_INITIALIZE</td>
<td>0x208</td>
<td>Rm</td>
<td>0</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Figure 3.3: (a) Visualization of the AIMC tile data-flow when using the ALPINE ISA extension. (b) The instruction definitions. To perform one matrix vector multiplication \(y = M \ast x\), first CM_QUEUE is called to send \(x\) to the AIMC tile’s input memory. Next, CM_PROCESS is called to perform \(y = M \ast x\), which takes the \(x\) sitting in the AIMC tile’s input memory, performs multiply-and-accumulate operations on each of the crossbar columns, and then stores the output \(y\) in the AIMC core’s output memory. Finally, CM_DEQUEUE is used to send \(y\) back to the CPU.

Interfacing Tightly-Integrated AIMC Tiles

In this section, I present the ISA extension for the tight AIMC integration. I implement the instructions using the previously unused opcodes in the ARMv8 architecture, as listed in Figure 3.3 (b). Prior to inference, the AIMC tile is programmed through the CM_INITIALIZE instruction, which writes 8-bit weight values to the indices of the AIMC crossbar. Active during the inference (our region of interest) are the other three instructions, which are utilized as follows.

The program packs 8-bit inputs into a 32-bit argument register. CM_QUEUE is then called to place the packed inputs into the input memory of the AIMC tile. Additional argument registers are used to specify the number of valid inputs packed in the aforementioned argument register, as well as the input memory index. Once all of the inputs are queued into the input memory, CM_PROCESS is called to operate the AIMC tile by converting the values from input memory into analog voltages via DACs, performing the MVM operation with the stationary weights, and storing the results in the output memory after digitizing them with ADCs. Finally, CM_DEQUEUE is called to retrieve packed 8-bit outputs from the AIMC output memory and place them in a destination register. The argument registers specify the number of packed outputs to retrieve and the index. A visualization of this process is in Fig. 3.3 (a).
```c
#include "aimclib.hh"

int main(int argc, char * argv[]) {
    // Mapping weights to the crossbar with x, y offset of 0, 0 using AIMClib.
    int8_t ** weights = ...;
    mapMatrix(0, 0, M, N, weights);

    for (int i = 0; i < N_INFERENCES; i++) {
        // Queue input array into the AIMC tile input memory using AIMClib.
        queueVector(sizeof(input) / sizeof(input[0]), input[i]);

        // Perform MVM using AIMClib.
        aimcProcess();

        // Dequeue output memory contents into output array using AIMClib.
        dequeueVector(N, output[i]);
    }

    return 0;
}
```

Figure 3.4: A sample C++ code for a single fully-connected feed-forward layer programmed onto AIMC tiles with AIMClib. At each inference step, the input is loaded and queued into the input memory. This is followed by the MVM via the `aimcProcess` method. Finally, the contents of the AIMC core output memory are dequeued straight into an output matrix data structure.

Interfacing Tightly-Integrated AIMC-Enabled Systems

The ISA extensions provide low-level support for tight AIMC integration, however I also developed a higher-level software library, AIMClib, to facilitate the development of AIMC-accelerated applications. This library is coded in C and can be used with C or C++ applications. In addition to containing the intrinsics in convenient in-lined wrapper methods (that use the C++ built-in `_asm`), it includes numerous functions and templates, such as the ability to queue/dequeue whole array or vector data structures of AIMC tile input/output memories, tiling matrices at offsets in the crossbar (so that multiple matrices of varying sizes can be placed next to each other), type-casting for tile inputs and outputs between `int8_t` and higher precision types (e.g. `fp32`), performing activation functions and other digital processing operations on tile outputs, and a checker program that simulates tightly-coupled AIMC tiles in guest software so that programs that utilize AIMClib can be debugged on the host machine before engaging the real or simulated hardware.

A sample of C++ pseudo-code using AIMClib is presented in Fig. 3.4. In addition to its own methods, I have also used AIMClib in conjunction with the Eigen C++ library (version 3.8). Eigen optimizes data structure space utilization and access as well as incorporates SIMD vector operations in our test applications [66].
3.3.3 AIMC Tile-Enabled System Simulation with ALPINE

Modeling AIMC Tiles in gem5-X

AIMC tiles are implemented in gem5-X using two classes: one class acts as a wrapper with metadata for setting up AIMC tiles, and the second class contains the AIMC tile itself. A visualization of these classes is depicted in Fig. 3.5.

The wrapper class is designed to encompass both the loose and tight coupling of AIMC tiles. For implementing the loose coupling, the wrapper is defined as a gem5 Peripheral Input/Output (PIO) device where it is accessible by load and store instructions at a specific memory-mapped address. Alternatively, for the tight coupling case, the wrapper can be accessed by dedicated instruction.

The AIMC tile class contains the actual crossbar and peripheral simulated modeling components. I base our AIMC tile implementation on [92]. Each generated AIMC tile has an input memory array, the memory crossbar, and output memory array. The component dimensions are parameterizable in the wrapper class configuration. Data conversions (ADCs/DACs) are simulated in software when the AIMC tile is interacted with (i.e., a MVM is performed by the AIMC tile).

AIMC-Enabled Systems in gem5-X

After the wrapper object and AIMC tile objects are defined, the next step is to instantiate them on the system and define the proper system interfaces (e.g., functional units and bus controller). This is realized in gem5-X systems through gem5-X’s configuration scripts, of which an example is found in Fig. 3.6. By itself, this allows for the system-level implementation...
of classic loosely-coupled AIMC-enabled systems that have permeated prior art by controlling how many AIMC tiles are generated for the entire simulated system and allowing access via bus and DMA interfaces.

To simulate the tightly-coupled AIMC-enabled architectures, I extend the accelerator modeling in [29] such that the custom ARMv8 ISA extension can also interface peripheral I/O (PIO) devices like our wrapper object. For this, I add connections between the ISA extension and PIO device via the system object (e.g., the simulated system that is instantiated on gem5-X’s launch). In our extension, the ISA templates also include the system object which is then subsequently included when the new instruction classes are generated during compilation. Furthermore, the wrapper class contains a pointer to the system that is set when the system first launches. The result is that when a custom instruction is called in a simulated program running in gem5-X full system mode, the custom instruction accesses a pointer to the system and then the wrapper object – at that point, the custom instruction can potentially choose which AIMC tile(s) to access and perform AIMC tile operations. The latency of the custom instructions is parameterizable, providing modeling flexibility on the AIMC tile. The return value of these instructions (the result held in the destination register) can also be data sent from the AIMC tile to the CPU. Furthermore, modeling of the sub-operations of the AIMC tile, including power-gating, analog-to-digital conversion, digital-to-analog conversion, and others, are included in the modeling of the latencies and operation of the custom ISA.

In our implementation for the initial exploration of tightly-coupled AIMC-enabled architectures, I generate one AIMC tile for each CPU core. Note that this is a design choice and the ALPINE framework supports alternative system definitions, including instantiating multiple AIMC tiles per CPU core, a hard-coded number of AIMC tiles, or others.

Figure 3.6: gem5-X Configuration Script code example for placing the AIMC wrapper class on the ARM SoC (RealView) platform. The instantiation is done in a similar manner for the loose-coupled and tight-coupled integration.
3.3.4 Tightly-Coupled AIMC Tile Exploration Setup and Methodology

Target Systems and Power Model

The system specifications of our gem5-X simulation are listed in Table 3.1-(A). I define two different system configurations to represent different use-cases, namely the high-power system such as those in higher-end devices and the low-power system, tailored for the embedded domain and internet-of-things edge devices. I would like to note that a substantial body of work focuses on AIMC tiles in high-power context; yet, the low stand-by power AIMC tile compels an exploration of its integration in low-power contexts as well [49], [99]. I use the MinorCPU model in our explorations, which is a 4-stage pipelined CPU with data forwarding and branch prediction [27].

The power models are shown in Table 3.1-(B). Our core and cache power model is based on a 28nm bulk system with an ARM Cortex A53 core [100], while our DRAM power model is based on [101]. The core and cache power model is comprised of active and WFM (wait for memory) CPU core energy per cycle, as well as energy/power for the last-level cache (LLC).

I calculate the total energy of the system using the gem5-X statistics. The generated statistics include total CPU cycles, simulated time, and cache/memory accesses for each experiment run. The full system energy is then the sum of the energies for the core, cache, and DRAM components.
### Table 3.1: ALPINE Tightly-Coupled AIMC Tiles Experimental Setup

#### (A) gem5-X FS Mode System Configurations

<table>
<thead>
<tr>
<th>System</th>
<th>Low-Power</th>
<th>High-Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Model</td>
<td>Minor (In-Order) CPU</td>
<td></td>
</tr>
<tr>
<td>Number of CPU Cores</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td>ARMv8 (AArch64)</td>
<td></td>
</tr>
<tr>
<td>CPU Core Frequencies</td>
<td>0.8GHz</td>
<td>2.3GHz</td>
</tr>
<tr>
<td>Supply voltage VDD</td>
<td>0.75V</td>
<td>1.3V</td>
</tr>
<tr>
<td>L1 Data/Instruction Cache Size</td>
<td>32kB</td>
<td>64kB</td>
</tr>
<tr>
<td>LLC Cache Sizes</td>
<td>512kB</td>
<td>1MB</td>
</tr>
<tr>
<td>Memory Model</td>
<td>8GB DDR4 @ 2400MHz</td>
<td></td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>16b</td>
<td></td>
</tr>
<tr>
<td>Memory Bus Frontend Latency</td>
<td>3 cycles</td>
<td></td>
</tr>
<tr>
<td>Memory Bus Forward, Response,</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>and Snoop Latencies</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### (B) System Energy and Power Figures

<table>
<thead>
<tr>
<th>System</th>
<th>Low-power</th>
<th>High-power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle Core Energy (pJ/Cycle)</td>
<td>10.72</td>
<td>126.03</td>
</tr>
<tr>
<td>WFM Core Energy (pJ/Cycle)</td>
<td>46.04</td>
<td>638.99</td>
</tr>
<tr>
<td>Active Core Energy (pJ/Cycle)</td>
<td>60.92</td>
<td>845.39</td>
</tr>
<tr>
<td>Mem Controller + IO Power (W)</td>
<td>3.03</td>
<td>5.82</td>
</tr>
<tr>
<td>LLC Leakage (mW/256kB)</td>
<td>271.62</td>
<td>874.08</td>
</tr>
<tr>
<td>LLC Read Energy (pJ/Byte)</td>
<td>1.81</td>
<td>5.60</td>
</tr>
<tr>
<td>LLC Write Energy (pJ/Byte)</td>
<td>1.63</td>
<td>5.02</td>
</tr>
<tr>
<td>DRAM Energy (pJ/Access)</td>
<td>120.0</td>
<td></td>
</tr>
</tbody>
</table>

#### (C) AIMC Tile Performance and Energy Figures

| AIMC tile crossbar size        | M rows, N columns |
| AIMC tile latency              | 100ns           |
| AIMC tile input/output data throughput | 4GB/s   |
| Input/output memory SRAM capacity | MB/NB |
| Supply voltage VDD (analog)    | 0.8V, 1.2V     |
| Supply voltage VDD (digital)   | 0.8V           |
| 256x256 AIMC tile MVM energy efficiency | 12.8 TOp/s/W |

* The AIMC tile MVM energy is re-calculated for varying tile sizes, considering the crossbar array size as well as data converters.
AIMC Setup and Modeling

Table 3.1-(C) reports the performance and energy metrics of the AIMC tile estimated from hardware measurements and chip designs in 14nm technology node [92], [102]. For compatibility with the core and cache model in 28nm node, I upscale the AIMC tile power estimates with a scaling factor of 5.3x for the high-power system and 2x for the low-power system. These factors are calculated following the classical scaling theory under constant frequency with the formulation \( \alpha \beta^2 \), where \( \alpha \) denotes the dimensional scaling and \( \beta \) is the voltage scaling factor [103].

Note that it is not a straightforward task to provide a simple power scaling factor for a mixed-signal design, such as our AIMC tile. One reason for this is the availability of different technology types or processes (e.g., high-performance, low-power, planar, FinFET) with specifications changing across foundries [104]. Secondly, digital and analog circuits follow different power scaling trends with the technology node [105]. Given that analog circuits scale less aggressively in comparison to its digital counterpart, our scaling represents a rather conservative estimate in this respect. I assume that the AIMC tile performance remains constant between the two technology nodes.

Exploration Studies Overview

To showcase ALPINE’s abilities and explore the benefits of tightly-coupling AIMC tiles in systems, I built and optimized a wide variety of neural network models using the Eigen C++ library and AIMClib. For multi-core experiments, I use libpthread to pipeline layers across cores, and implement ping-pong buffering to prevent input/output blocking [66].

With these tools in hand, I then perform three neural network explorations. First, I consider a Multi-Layer Perceptron (MLP) to gauge the performance and energy benefits of a neural network that almost solely relies on matrix-vector multiplication (MVM) operations. Then I consider an alternative neural network architecture, the Long Short-Term Memory (LSTM), that has increased computational requirements outside the MVM. Finally, I look at the fully-pipelined implementation of a Convolutional Neural Network (CNN) and explore how the proposed system behaves in the presence of large number of MVMs in conjunction with very intense memory access patterns. I further breakdown these applications into multiple cases with different AIMC tile and CPU core mappings. All of these AIMC tile-enabled neural networks and their implementations are compared against a digital-only, SIMD-enabled, reference application, which employs the same aforementioned optimizations (Eigen Library integration, pthreads, ping-pong buffers). Furthermore, for more equitable performance comparisons, I use similar precision across all applications (\( \text{int8}_t \) with \( \text{fp32} \) accumulation where floating point operations apply, such as in sigmoid and softmax operations).

In general, inference-to-inference I notice a deviation in performance results and system metrics of less than 4%. Thus, to constrain simulation time, I only perform 10 inferences
for each of the cases in the MLP and LSTM neural networks. I further reduce the number of inferences in the CNNs to 3 due to the larger network requiring more simulation time.

### 3.3.5 Multi-Layer Perceptron Mappings and Analyses

#### Multi-Layer Perceptron Architecture and Cases

In this first case study, I focus on a two-layer MLP neural network (1024 by 1024) with ReLU activation functions (Figure 3.7 (a)). While this acts as a simple example, it is worth noting that MLP-based implementations are becoming more prevalent through recommender systems [106] and vision [107]. Furthermore, Google reported that it is still a significant part of their workloads [71].

Because the activation functions are computationally simple, the main component of the MLP run-time (without the use of the AIMC cores) is the matrix-vector multiplication (MVM) via the multiply-and-accumulate (MAC) operations in the two fully-connected layers. The AIMC tiles specifically speed up these operations, so I can gauge the maximum performance benefits attainable by leveraging the AIMC cores.

I create four different analog MLP implementations where vary the size and number of AIMC tiles, as shown in Figure 3.7 (b). More specifically, Cases 1 and 2 are single-CPU core architectures that use one large AIMC tile. Case 3 is a dual-CPU core architecture with one fully-connected layer assigned to each CPU core. Each AIMC tile is also smaller in capacity relative to the previous cases. Finally, Case 4 is a quad-CPU core architecture with one fully-connected layer’s computation being split between two CPU cores. The two CPU cores of the first layer sync their outputs via mutexes before letting the second layer start its processing. Additionally, I compare these implementations with a conventional CPU-only and SIMD-enabled architecture.
MLP Single-Core Results and Analysis

Figure 3.8: Aggregate results for multi-layer perceptron experiments. From left to right, each column contains total time, memory intensity, and energy results for the High-Power system (top row) and Low-Power system (bottom row) configurations. "ANA" refers to analog AIMC-enabled application mappings with implementation numbers corresponding to those in Fig. 3.7, while "DIG" refers to a digital reference or CPU-only implementation. Results are also grouped by the number of CPU cores utilized (1, 2, or 4).

<table>
<thead>
<tr>
<th>Case</th>
<th>Runtimes</th>
<th>Memory Intensity</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DIG</td>
<td>ANA</td>
</tr>
<tr>
<td>HP Single-Core (1)</td>
<td>0.0295s</td>
<td>0.0023s</td>
<td>0.056</td>
</tr>
<tr>
<td>HP Single-Core (2)</td>
<td>0.0295s</td>
<td>0.0023s</td>
<td>0.056</td>
</tr>
<tr>
<td>HP Dual-Core (3)</td>
<td>0.0144s</td>
<td>0.0028s</td>
<td>0.029</td>
</tr>
<tr>
<td>HP Quad-Core (4)</td>
<td>0.0093s</td>
<td>0.0030s</td>
<td>0.030</td>
</tr>
<tr>
<td>LP Single-Core (1)</td>
<td>0.0487s</td>
<td>0.0059s</td>
<td>0.058</td>
</tr>
<tr>
<td>LP Single-Core (2)</td>
<td>0.0487s</td>
<td>0.0061s</td>
<td>0.058</td>
</tr>
<tr>
<td>LP Dual-Core (3)</td>
<td>0.0258s</td>
<td>0.0073s</td>
<td>0.040</td>
</tr>
<tr>
<td>LP Quad-Core (4)</td>
<td>0.0175s</td>
<td>0.0078s</td>
<td>0.041</td>
</tr>
</tbody>
</table>

Table 3.2: Aggregate MLP Results

The aggregate results for all of our MLP experiments are shown in Figure 3.8 and Table 3.2. I focus on the run-time, energy footprint and memory intensity (quantified as last-level cache misses per instruction, or LLCMPI). The latter provides important insights onto the data movement overhead, which is a significant driver for AIMC-based acceleration.

In all implementations, AIMC tiles provide significant benefits in terms of latency and energy in comparison to the CPU-only runs. Looking at the single-CPU implementations, I observe that Case 1 out-performs Case 2 by a slight margin in terms of latency and energy. While a similar amount of queuing and dequeuing takes place in both implementations, the CM_PROCESS instruction needs to be called twice as much in order to perform the same number of inferences in Case 2. Yet, this does not translate into a 2x run-time and energy-overhead, because the MVMs account for only a small fraction of the total run time, as shown in Figure.
Figure 3.9: The run-time analysis of MLP cases. Run-time percentage of each sub-ROI division in MLP cases averaged across high-power and low-power systems. The \textit{average reference} is obtained across single-, dual-, and quad-core digital cases. \textit{Non-MVM Digital Operations} refers to the combined run times of the input load, digital activation, and output writeback. The standard deviation of the timing distribution is less than 1.7\%, 1.2\%, 2.3\%, and 1.5\% for the average reference, Case 1, Case 3, and Case 4, respectively. The time distribution of cases 1 and 2 are similar and hence, Case 2 is omitted for brevity.

3.9. Because of the magnitude of the analog queue/dequeue times on total run time shown in this figure, I show that it is critical to provide a sufficiently large queue/dequeue bandwidth to maximize the benefits from AIMC acceleration, as also discussed in [49].

In addition to the experiments showcased in Figure 3.7, I also tested a "loosely-coupled" AIMC tile-enabled system that places two pipelined AIMC tiles with dedicated ReLU activation units next to each other in an off-chip accelerator. A single CPU core handles data transactions like sending inputs and receiving outputs from the MLP, which is mapped to the two AIMC tiles. While this configuration attains 4.1x speedup over the single-core digital reference MLP, it has up to a 3.1x slowdown when compared to the tightly-coupled AIMC-enabled system. Loosely-coupled AIMC tile-enabled architectures will be explored more deeply in Chapter 4.

MLP Multi-Core Results and Analysis

The latency and energy for the multi-core Cases 3 and 4 are displayed in Figure 3.8, as well as their time distribution in Figure 3.9. What results make apparent with respect to the single-CPU core implementations is that adding multiple CPU cores does not equate to more performance gains. In fact, the performance and energy of the system worsens with increasing
number of CPU cores: the single CPU-core Case 1 has approximately 20% and 30% better run
time over the dual CPU-core Case 3 and quad CPU-core Case 4, respectively.

This slowdown is mainly attributed to the communication overhead of sending inputs and
activations across CPU cores. As the number of CPU cores increase, I observe that the run-
time associated with \textit{input load} and \textit{analog queue} make up a larger portion of the overall run-time (Figure 3.9). More specifically, in the dual-CPU core implementation Case 3, the
overhead of communicating the output from the first layer to the input of the second layer
significantly increases the total run time of the application. The overhead is then compounded
in the quad-CPU core implementation Case 4 where the input from memory as well as the
intermediate activations must be sent to two different CPU cores. The synchronization
overhead associated with mutexes of both layers aggravates this, as well. Finally, it is worth
noting that the performance impact of the CM\_PROCESS operation becomes negligible in the
AIMC tile-enabled algorithms, so even estimates of the latency increased 10x are observed to
have minimal impact on the performance results.

Interestingly, the memory intensity remains almost constant across all implementations
(Cases 1 through 4), suggesting that memory access is no longer a significant bottleneck.
Instead, I conclude that for an application whose run-time is dominated by \textit{input load} and
\textit{analog queue}, the overhead of sharing data is no longer negligible and should be treated as
the primary bottleneck to gains in run time and energy.

Finally, across all cases, the low-power system exhibits lower performance gains in comparison
to the high-power system. This is primarily due to the smaller L1 cache size of the low-power
system configuration. A smaller cache requires more requests (and, therefore, endures more
delay) to the L2 cache and memory, which is reflected in the slightly higher memory intensity
metric relative to the high-power system.

**MLP Computational Complexity**

In this section, I present a computational complexity analysis for the CPU-only and AIMC-
based implementations. I will assume that the limited cache size and cache trashing, as well
as the SIMD operations, do not impact the computational complexity. For the CPU-only run,
each fully-connected layer's MVM operation has a quadratic complexity ($O(n^2)$), while the
 corresponding activation function (ReLU), as well as loading initial inputs (\textit{input load}) from
memory and storing outputs (\textit{output writeback}), has a linear complexity ($O(n)$). Given that
the MLP experiments run for $N_{inf}$ inferences, the total complexity of the MLP run can be
formulated as $N_{inf} \times (2O(n^2) + 4O(n)) \approx O(N_{inf}n^2)$.

With the introduction of the AIMC tiles however, the complexity of MVM operations reduces
to constant time ($O(1)$), assuming that the entirety of the weights of the fully-connected layer
can fit in the AIMC tile. Therefore the total computational intensity reduces to $N_{inf} \times (2O(1) +
6O(n)) \approx O(N_{inf}n)$ after including complexities for analog queueing, shifting the dominating
run-time factor to the linear operations (queuing inputs/dequeuing outputs; Fig. 3.9).

MLP Compute Requirements

In this section, I analyze the memory footprint for the CPU-only and AIMC-based implementations. In the CPU-only implementation, the weights of the fully-connected layers must be loaded from the main memory into L1/L2 caches at every inference. Yet, this is not true in the analog implementations.

Let us define the working set as the required amount of data memory per inference. In the CPU-only implementation, this includes the weights of the fully connected layers (2W), the inputs loaded from memory (x), the intermediary activations (l1), and the final outputs stored to memory (y). For our implementation with 8-bit weights, inputs, activations and outputs, the working set size is 2W + x + l1 + y = 2 * n^2 + 3n ≈ 2.1MB for n = 1024. For all of our experimental configurations, this working set size exceeds that of both the private L1 caches and the shared L2 cache, meaning elements of the fully-connected layers and the input/output must be thrashed (swapped in and out of the caches, as well as potentially main memory), leading to both worse memory performance and worse total run time.

In contrast, the AIMC-enabled MLP keeps all of the weights of the fully-connected layers stationary inside the AIMC tiles. After the one-time cost for programming the weights in our MLP, the weights are never reprogrammed, and therefore, can effectively be removed from the working set. In this case, the working set size can be formulated as x + l1 + y = 3n ≈ 3kB, which fits comfortably in L1 private caches for both of our test system configurations. This leads to lower memory intensity, less cache thrashing, and thus improved overall performance.

The reduction in computation and throughput requirements resulting from the introduction of the AIMC tiles in the single-CPU core cases 1 and 2 is related to the ones obtained with multi-threading (cases 3 and 4), with the caveat that the computational complexity goes down by the number of hardware threads and space complexity is distributed across the CPU cores. However, even though the space complexities are reduced, the impact of the linear computational complexities of input load and analog queue are increased by the emerging core-to-core communications bottleneck in the multi-CPU core applications (cases 3 and 4).

I therefore reiterate that when AIMC tiles are introduced to neural networks with very small digital operation requirements (such as only ReLU activation functions), that core-to-core communications overhead should be minimized by using fewer CPU cores and AIMC tiles. By distributing simple digital computation across numerous CPU cores, core-to-core communication emerges as the new bottleneck and can hinder, rather than help, the performance of multi-core-enabled neural networks.
3.3.6 Recurrent Neural Network Mappings and Analyses

**LSTM Architecture**

In the previous section, I examined how the AIMC tiles can benefit applications where the primary overhead is the MVM operation. However, another perspective one can take of tightly-integrated AIMC tiles is that they are a co-processor with respect to the CPU, which can help to decrease the communications latency of interacting with digital logic. Furthermore, tightly-integrated AIMC tiles allow us to leverage preexisting hardware for operations that would be expensive (either with respect to area or communications latency) to perform in a loosely-coupled configuration. And thus in our second exploration, I look at recurrent neural networks (RNNs) in the form of a Long-Short Term Memory (LSTM) network targeting character recognition using the Penn Treebank (PTB) data set [108]. The LSTM has one cell (hidden) layer and one fully-connected (dense) layer, as presented in Figure 3.10 (a). In comparison to the MLP, the LSTM features more computationally heavy digital operations (sigmoid, tanh, softmax). Moreover, the data flow bears differences owing to the recurrent connection of the LSTM cell. Therefore, the LSTM allows for the evaluation of performance benefits of leveraging AIMC tiles when digital operations play a larger role in overall execution.

![LSTM Network Architecture](image)

Figure 3.10: (a) The neural network diagram for the LSTM modeled in our AIMC-enabled test programs. It is a 2-layer LSTM with one LSTM hidden cell layer and one dense (fully-connected) layer with various, more compute-intensive, activation functions. Note that the AIMC sizes are variable depending on the size of the hidden cell layer and that all activation functions are performed in the CPU cores. (b) Cases 1 through 4 show the mappings of the layers to a variety of AIMC-based configurations.
Table 3.3: LSTM Experiment Setup

(A) LSTM Neural Network Parameters

<table>
<thead>
<tr>
<th>Input (x)</th>
<th>Hidden Layer ($n_h$)</th>
<th>Output (y)</th>
<th>Total Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>256</td>
<td>50</td>
<td>377.3k</td>
</tr>
<tr>
<td>50</td>
<td>512</td>
<td>50</td>
<td>1.28M</td>
</tr>
<tr>
<td>50</td>
<td>750</td>
<td>50</td>
<td>2.6M</td>
</tr>
</tbody>
</table>

(B) LSTM AIMC Tile Dimensions

<table>
<thead>
<tr>
<th>$n_h$</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>612 x 1074</td>
<td>356 x 1074</td>
<td>356 x 1024</td>
<td>356 x 256</td>
</tr>
<tr>
<td>512</td>
<td>1124 x 2098</td>
<td>612 x 2098</td>
<td>612 x 2048</td>
<td>612 x 512</td>
</tr>
<tr>
<td>750</td>
<td>1600 x 3050</td>
<td>850 x 3050</td>
<td>850 x 3000</td>
<td>850 x 750</td>
</tr>
</tbody>
</table>

Figure 3.10 (b) shows the different simulated cases. Cases 1 and 2 are single-CPU core cases that use larger AIMC cores. Case 3 is a dual-CPU core case with the cell layer assigned to the first CPU core, and the dense layer assigned to the second CPU core. Finally, Case 4 is a quin-CPU core case with the cell layer’s computation split across the first four CPU cores, and the dense layer assigned to the last CPU core. Here, the four CPU cores associated with the LSTM cell synchronise their outputs via mutexes before the second layer starts its MVM operation.

To evaluate the decreased computational complexity hypothesized and elaborated upon in the first case study, I focus on LSTM instances sharing the same architecture but with different layer sizes. The dimensions of the layers in these networks, as well as the corresponding AIMC tile sizes, are listed in Table 3.3.

To reduce CPU core-to-core communication as much as possible in case 4, the LSTM cell layer is mapped to AIMC tiles such that instead of the gates being distributed to different AIMC tiles, they are sliced so that element-wise operations can be performed by reading four consecutive columns [102].

Note that the largest LSTM architecture ($n_h = 750$) is shown to experimentally yield high accuracy when implemented on real PCM prototype hardware chip [95].

**LSTM Single Core Results and Analysis**

Aggregate results for all of our LSTM experiments are shown in Figure 3.11 and Table 3.4, including multi-core results, results for both system configurations, and factor improvements with the largest of the networks ($n_h = 750$). When $n_h$ is 256, I observe 1.0-1.5x factor improve-
Chapter 3  Tightly-Coupled AI Accelerator Solutions

Figure 3.11: Aggregate results for all LSTM experiments. From left to right, each column contains total time, memory intensity, and energy results for the high-power system (top row) and low-power system (bottom row) configurations. "DIG" refers to digital reference applications while "ANA" refers to a specific analog, AIMC-enabled application case, which correspond to those in Figure 3.10. Results are grouped by the number of CPU cores utilized, and from left to right, each grouped column refers to a different $n_h$ parameter, which affects the total size of the network. The darker bars refer to a larger $n_h$.

Figure 3.12: Bar graphs showing the percentage of the ROI run time going towards sub-ROIs for the analog cases 1 through 4 run on the high-power system configuration for all $n_h$ values. The trends of the high-power system sub-ROI run times are followed by the same cases in the low-power system configuration. The digital cases (single-, dual-, and quin-CPU core) see 87.8% to 97.9% of their total ROI run time dedicated to the digital MVM operations with activation functions in the cell layer. The "Misc." run time is comprised of all other sub-ROIs per experiment that comprise of less than 5% of the total ROI run time.
### A. Runtime Results

<table>
<thead>
<tr>
<th>Case</th>
<th>$n_h = 256$</th>
<th></th>
<th>$n_h = 512$</th>
<th></th>
<th>$n_h = 750$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
<td>ANA</td>
</tr>
<tr>
<td>HP Single-Core (1)</td>
<td>0.0038s</td>
<td>0.0027s</td>
<td>0.0123s</td>
<td>0.0034s</td>
<td>0.0361s</td>
<td>0.0038s</td>
</tr>
<tr>
<td>HP Single-Core (2)</td>
<td>0.0038s</td>
<td>0.0027s</td>
<td>0.0123s</td>
<td>0.0035s</td>
<td>0.0361s</td>
<td>0.0040s</td>
</tr>
<tr>
<td>HP Dual-Core (3)</td>
<td>0.0411s</td>
<td>0.0311s</td>
<td>0.0118s</td>
<td>0.0038s</td>
<td>0.0221s</td>
<td>0.0044s</td>
</tr>
<tr>
<td>HP Quin-Core (4)</td>
<td>0.0033s</td>
<td>0.0031s</td>
<td>0.0056s</td>
<td>0.0033s</td>
<td>0.0086s</td>
<td>0.0035s</td>
</tr>
<tr>
<td>LP Single-Core (1)</td>
<td>0.0106s</td>
<td>0.0071s</td>
<td>0.0338s</td>
<td>0.0090s</td>
<td>0.0632s</td>
<td>0.0103s</td>
</tr>
<tr>
<td>LP Single-Core (2)</td>
<td>0.0106s</td>
<td>0.0072s</td>
<td>0.0338s</td>
<td>0.0094s</td>
<td>0.0632s</td>
<td>0.0107s</td>
</tr>
<tr>
<td>LP Dual-Core (3)</td>
<td>0.0113s</td>
<td>0.0082s</td>
<td>0.0304s</td>
<td>0.0105s</td>
<td>0.0401s</td>
<td>0.0118s</td>
</tr>
<tr>
<td>LP Quin-Core (4)</td>
<td>0.0091s</td>
<td>0.0082s</td>
<td>0.0148s</td>
<td>0.0089s</td>
<td>0.0175s</td>
<td>0.0094s</td>
</tr>
</tbody>
</table>

### B. Memory Intensity (LLCM/Instruction) Results

<table>
<thead>
<tr>
<th>Case</th>
<th>$n_h = 256$</th>
<th></th>
<th>$n_h = 512$</th>
<th></th>
<th>$n_h = 750$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
<td>ANA</td>
</tr>
<tr>
<td>HP Single-Core (1)</td>
<td>0.0071</td>
<td>0.0069</td>
<td>0.0179</td>
<td>0.0052</td>
<td>0.0366</td>
<td>0.0045</td>
</tr>
<tr>
<td>HP Single-Core (2)</td>
<td>0.0071</td>
<td>0.0063</td>
<td>0.0179</td>
<td>0.0045</td>
<td>0.0366</td>
<td>0.0041</td>
</tr>
<tr>
<td>HP Dual-Core (3)</td>
<td>0.0081</td>
<td>0.0066</td>
<td>0.0199</td>
<td>0.0048</td>
<td>0.0443</td>
<td>0.0045</td>
</tr>
<tr>
<td>HP Quin-Core (4)</td>
<td>0.0069</td>
<td>0.0053</td>
<td>0.0187</td>
<td>0.0047</td>
<td>0.0384</td>
<td>0.0040</td>
</tr>
<tr>
<td>LP Single-Core (1)</td>
<td>0.0110</td>
<td>0.0102</td>
<td>0.0365</td>
<td>0.0077</td>
<td>0.0381</td>
<td>0.0067</td>
</tr>
<tr>
<td>LP Single-Core (2)</td>
<td>0.0110</td>
<td>0.0097</td>
<td>0.0365</td>
<td>0.0070</td>
<td>0.0381</td>
<td>0.0062</td>
</tr>
<tr>
<td>LP Dual-Core (3)</td>
<td>0.0135</td>
<td>0.0102</td>
<td>0.0367</td>
<td>0.0076</td>
<td>0.0468</td>
<td>0.0067</td>
</tr>
<tr>
<td>LP Quin-Core (4)</td>
<td>0.0151</td>
<td>0.0088</td>
<td>0.0327</td>
<td>0.0072</td>
<td>0.0403</td>
<td>0.0065</td>
</tr>
</tbody>
</table>

### C. Energy Results

<table>
<thead>
<tr>
<th>Case</th>
<th>$n_h = 256$</th>
<th></th>
<th>$n_h = 512$</th>
<th></th>
<th>$n_h = 750$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
<td>ANA</td>
</tr>
<tr>
<td>HP Single-Core (1)</td>
<td>0.016J</td>
<td>0.011J</td>
<td>0.050J</td>
<td>0.014J</td>
<td>0.147J</td>
<td>0.016J</td>
</tr>
<tr>
<td>HP Single-Core (2)</td>
<td>0.016J</td>
<td>0.011J</td>
<td>0.050J</td>
<td>0.014J</td>
<td>0.147J</td>
<td>0.016J</td>
</tr>
<tr>
<td>HP Dual-Core (3)</td>
<td>0.017J</td>
<td>0.012J</td>
<td>0.049J</td>
<td>0.016J</td>
<td>0.092J</td>
<td>0.018J</td>
</tr>
<tr>
<td>HP Quin-Core (4)</td>
<td>0.015J</td>
<td>0.014J</td>
<td>0.029J</td>
<td>0.015J</td>
<td>0.046J</td>
<td>0.016J</td>
</tr>
<tr>
<td>LP Single-Core (1)</td>
<td>0.006J</td>
<td>0.004J</td>
<td>0.021J</td>
<td>0.005J</td>
<td>0.039J</td>
<td>0.006J</td>
</tr>
<tr>
<td>LP Single-Core (2)</td>
<td>0.006J</td>
<td>0.004J</td>
<td>0.021J</td>
<td>0.006J</td>
<td>0.039J</td>
<td>0.006J</td>
</tr>
<tr>
<td>LP Dual-Core (3)</td>
<td>0.007J</td>
<td>0.005J</td>
<td>0.019J</td>
<td>0.006J</td>
<td>0.026J</td>
<td>0.007J</td>
</tr>
<tr>
<td>LP Quin-Core (4)</td>
<td>0.006J</td>
<td>0.005J</td>
<td>0.011J</td>
<td>0.005J</td>
<td>0.014J</td>
<td>0.006J</td>
</tr>
</tbody>
</table>

Table 3.4: Aggregate LSTM Results
ments across all metrics and system configurations in the AIMC tile-enabled cases over the digital case due to the very small working set. However, when \( n_h \) increases to 512 and then 750, the run time and energy of the digital application increases up to 9.4x/9.3x with the working set size increase of 7x. This is in comparison to the AIMC tile-enabled applications, which sees an average run time and energy increase of 1.4x, suggesting a sub-quadratic increase in run-time complexity with higher space complexity.

Relative to the first case study with the MLP, I see smaller maximum time and energy gains (9.4x/9.3x versus 12.8x/12.5x); this is expected as a greater proportion of the LSTM total run-time is dedicated to digital operations that do not see a reduced computational complexity with the introduction of the AIMC tile. However, these digital operations do see mild performance improvements as a result of lower memory intensity and therefore less cache thrashing (due to space freed from lack of weights loaded in AIMC-enabled LSTMs).

### LSTM Multi-Core Results and Analysis

Similar to the single-CPU core cases, the multi-CPU core cases also have significant performance gains as \( n_h \) grows larger, in comparison to the digital implementation (Figure 3.11). By examining the sub-Region of Interest (ROI)s of the LSTM inference, as seen in Figure 3.12, I see that the new bottleneck of the LSTM algorithm is the cell layer dequeuing and activation functions (up to 81.8% of the inference run time), followed by the cell gate combinations (up to 14.9% of the inference run time). After more analysis, the activation functions in the cell layer alone account for approximately 70% of the cell dequeue and activation run time, meaning that the cell layer's digital component in the AIMC tile-enabled systems accounts for up to 57.3% of the algorithm's run-time. Hence, unlike with the MLP study, going from single-CPU core to multi-CPU core with the LSTM in the AIMC tile-enabled implementations does result in a speedup of 10% (cases 1 vs. 4), due to the LSTM cell's parallelized linear operations (but not more due to inter-layer communication).

### LSTM Complexity

In the digital reference application, the computation of the LSTM cell outputs involve four MVM operations of quadratic complexity (\( 4O((n_h) \ast (x + n_h)) \approx 4O(n^2) \)) and nine operations of linear complexity (sigmoid, hyperbolic tangent, array multiplication, array addition; \( 9O(n_h) \approx 9O(n) \)). In addition to the LSTM cell's added complexity, the softmax operation is used as the fully-connected layer's activation function which doubles in complexity when compared to ReLU \( 2O(y) \approx 2O(n) \). This is in addition to loading inputs and storing results \( O(x) + O(y) \approx 2O(n) \). Thus the total computational complexity of the ROI of our LSTM application for \( N_{inference} \) inferences is \( N_{inference} \ast (5O(n^2) + 13O(n) = O(N_{inference} n^2) \). Even though the computational complexity of the LSTM is similar to the MLP, it is actually more than 3x more intensive than the MLP in terms of linear digital operations and contains three more MVM operations per inference.
With the introduction of the AIMC tiles in the single- and dual-CPU core cases however, the application queues the concatenated input \([h_{t-1}, x]\) into the AIMC tile's input memory for the LSTM cell, and then perform all four MVM operations with one CM_PROCESS instruction call by tiling the LSTM cell weights next to each other (in cases 1 and 2). Then when the application fetches the result, it is actually fetching the concatenated MVM results of the forget, input, activation, and output gates (MVM results using \(W_f\), \(W_i\), \(W_a\), and \(W_o\), respectively) before the activation functions are performed. Thus the computational complexity of the LSTM cell layer reduces the \(4O(n^2)\) factor to \(O(1)\) while adding the queuing and dequeuing complexities of \(O([h_{t-1}, x]) + O(n_h) \approx 2O(n)\). Subsequently, the total computational complexity can be reduced to \(N_{inf} * (2O(1) + 15O(n)) \approx O(N_{inf}n)\), or linear complexity. The reduced complexity explains why as \(n_h\) increases there is not a substantial increase in the run time of the AIMC-enabled LSTM application, as opposed to non-accelerated (digital reference) mappings.

Similar to the MLP, when the layers are pipelined together in the multicore cases the complexity of both the AIMC-enabled and reference applications reduces by the number of hardware threads used. Unlike the MLP however, there is not a slowdown as a result of taking our AIMC-tiled application multicore, due to the increased number of digital operations in the LSTM cell being more adequately split across cores.

### LSTM Working Set Analysis and Space Complexity

Just like with the MLP application however, the LSTMs also make significant gains in memory intensity by using stationary weights, and thus reducing the size of the working set. If I calculate the approximate working set of the LSTM application, it is comprised of the input \((x + h_0)\), the LSTM cell layer's weights \((4 * (n_h * (n_h + x)))\), the fully-connected layer's weights \((n_h * y)\), the intermediary result in-between the layers \((n_h)\), and the output \((y)\). The total size complexity of the LSTM application per inference using 8-bit types is \((x + n_h) + 4(n_h^2 + n_hx) + n_h + n_h y + y\). Using the numbers from Table 3.3, the size of the working set for \(n_h = 256\), 512, 750 is 378kB, 1.28MB, and 2.59MB, respectively. Even in the smallest variant of the LSTM application, the working set cannot entirely fit in the private caches of the CPU core(s). When \(n_h\) is 512 or 750, in both low-power and high-power system configurations, the CPU core(s) must go out to L2 cache and main memory to contain the working set.

When I utilize AIMC tiles in our LSTM application, the weights are removed from the working set because they are never needed by the CPU core(s) during the ROI, thus reducing our total size complexity to \((x + n_h) + n_h + y\), which is 0.66kB, 1.17kB, and 1.65kB for \(n_h = 256\), 512, and 750, respectively. For all values of \(n_h\) tested, the working set can fit entirely in L1 private caches for both low-power and high-power system configurations, hence the increasing performance improvements and lower memory intensity with greater values of \(n_h\).

Therefore, while AIMC tiles greatly speedup neural networks where MVMs are the dominating operation, careful attention must be paid to both the size of the neural network and the proportion of other non-optimized digital operations. This case study of the LSTM shows
that when linear operations are more dominant, that realized gains in performance are lower with the AIMC tiles (as compared with the first exploration study), and thus these other linear operations which are not optimized with the inclusion of the AIMC tiles become the new bottleneck. Furthermore, when the neural network is small enough to efficiently leverage a CPU core’s L1 private cache and a small portion of L2 cache (as when \( n_h \) is 256), the overhead of queuing inputs and dequeuing outputs to and from the AIMC tile ends up having a similar run-time (with only minor performance gains) to the reference application, which otherwise invalidates or diminishes the potential benefit of introducing AIMC tiles.

3.3.7 Convolutional Neural Network Mappings and Analyses

CNN Architecture

Figure 3.13: (a) The architecture of the CNNs presented in [109] and their mapping onto the ALPINE systems. The blue boxes with the AIMC tiles represent the convolutional layers. The dense layers are not mapped to AIMC tiles. (b) shows the dimensions and parameters of each CNN. The CNN has 5 convolutional layers (3 with Max Pooling), 3 dense layers, and ReLU activation functions for all layers except the last layer, which uses Softmax.
For our last exploration study, I explore the benefits of introducing AIMC tiles for CNNs in an 8-CPU core MPSoC. Contrary to the previous studies, where each layer weights are used only once per inference, convolution operations require multiple passes on weights per inference via shifting kernels over the feature maps. To perform the convolution operations in AIMC tile-enabled applications, I flatten the kernels into columns and store these in the columns of the AIMC tile, as described in [77], [110]. The feature maps are also flattened and queued to the AIMC tile.

I explore the three CNN variants presented in [109] to act as a baseline for modern CNNs; they are labeled CNN-F(ast), CNN-M(edium), and CNN-S(low). While CNN-S and CNN-M are similarly sized, the increase of the MaxPool operation factor in layers 1 and 5 from x2 to x3 increases the computational requirements of CNN-S significantly by performing strided 3x3 pool operations instead of 2x2 pool operations. Figure 3.13 (a) shows the proposed CNN implementation and data flow, while Figure 3.13 (b) reports the CNN architecture parameters. Fine-grained pipelining is applied for the data-flow; the convolutions are performed whenever the corresponding input volume of the feature map is available. Contrary to the previous exploration studies, I utilize the AIMC tiles only for convolutional layers. The feed-forward layers are processed in the CPU; these layers are executed only once as opposed to the convolutional layers and therefore do not constitute a bottleneck.

CNN Results and Analysis

![Figure 3.14: Aggregate results for CNN experiments. From left to right, each column contains total time, memory intensity, and energy results for the High-Power system (top row) and Low-Power system (bottom row) configurations. ANA refers to analog AIMC-enabled applications with CNN names corresponding to those in Table 3.13, while DIG refers to a digital reference, non-AIMC-enabled, implementation. The CNNs F, M, and S represent fast, medium, and slow variations on the same CNN architecture, respectively.]

I present our results in Figure 3.14. The largest performance increase with respect to the CPU-only implementation is recorded for the largest CNN variant “S”. This configuration exhibits the maximum speedup of 20.5x, a memory intensity improvement of 3.7x, and an energy improvement of 20.8x for the high-power system.
### Table 3.5: Aggregate CNN Results

<table>
<thead>
<tr>
<th>Case</th>
<th>Runtimes</th>
<th>Memory Intensity</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIG</td>
<td>ANA</td>
<td>DIG</td>
</tr>
<tr>
<td>HP Chatfield-F</td>
<td>5.09s</td>
<td>0.70s</td>
<td>0.014</td>
</tr>
<tr>
<td>HP Chatfield-M</td>
<td>16.67s</td>
<td>1.34s</td>
<td>0.015</td>
</tr>
<tr>
<td>HP Chatfield-S</td>
<td>23.72s</td>
<td>1.15s</td>
<td>0.016</td>
</tr>
<tr>
<td>LP Chatfield-F</td>
<td>13.09s</td>
<td>1.48s</td>
<td>0.016</td>
</tr>
<tr>
<td>LP Chatfield-M</td>
<td>38.20s</td>
<td>3.41s</td>
<td>0.016</td>
</tr>
<tr>
<td>LP Chatfield-S</td>
<td>52.81s</td>
<td>3.00s</td>
<td>0.017</td>
</tr>
</tbody>
</table>

Figure 3.15: CPU utilization for CNN-S in the high-power system, expressed as the percentage of idle CPU cycles (top) and instructions/cycle (IPC) (bottom).

While in the prior case studies the core utilization across layers does not vary widely across experiments, the CNN benchmark is used to examine the AIMC acceleration with uneven CPU core utilization. To this end, Figure 3.15 shows both the CPU idle percentage and the instructions per cycle (IPC) count for each individual CPU core in our high-power CNN-S application. CNN-M and CNN-F in both low-power and high-power system configurations exhibit very similar trends. The utilization of the first convolutional layer is similar in both CPU-only and AIMC tile-enabled benchmarks due to input load from memory. For convolutional layers 2 and 3, AIMC tiles provides significant benefits with idle cycles decreasing up to 4x. Likewise, IPC increases relative to the CPU-only benchmark by up to 3x. Convolutional layers 4 and 5 exhibit more idle cycles in the AIMC-based implementation in comparison to prior layers. This is partly attributed to reduced size of the feature maps, owing to the stride and pooling operations of the previous layers. The fully-connected layers’ CPU cores spent the most time idling as fully-connected layers are utilized once during inference.

As it is already established that the memory intensity of the AIMC-enabled application goes down significantly, the drop in IPC for these last two convolution layers is direct evidence of a communications and computational bottleneck. Further evidence comes from how the fully-connected layers (which are not accelerated) do not incur a speedup and subsequently do not see an increase in IPC, despite more system resources (namely, L2 shared cache) being
made available by the lack of CNN kernels in the working set in the AIMC tile-enabled CNN.

Owing to the fine-grained activation pipelining, the amount of data to be communicated between layers are significantly reduced. Yet, the total inference run-time is more than the MLP and LSTM cases owing to the multiple passes over the convolutional kernels.

As a result, while the AIMC tiles offer significant speedups for CNNs as well, further exploration is needed to optimize the data flow for shifting bottlenecks in layers. This includes replicating the initial layer convolutional kernels to balance the CNN pipeline owing to varying feature map sizes [111], including local SRAM in the AIMC tile to avoid queueing the same input volume of the feature maps multiple times [111] and minimizing core-to-core communication overheads. All these explorations can be carried out on the ALPINE framework; strategies for approaching load balancing in CNNs using AIMC tiles are explored in Chapter 4.

### 3.3.8 Tightly-Coupled Analog In-Memory Compute Tiles: Key Takeaways and Conclusions

In this exploration, I presented and explored the performance benefits of a novel architecture that utilizes tightly-coupled AIMC tiles. Using ALPINE, I extended the ARMv8 ISA with custom instructions that interface AIMC tiles directly from their execution in the CPU. Using ALPINE and AIMClib, I then implemented and tested three different exploration studies across two system configurations, namely, single and multi-core MLPs, single and multi-core LSTMs, and finally multi-core CNNs. Through these exploration studies I observed how computational and size complexity is reduced by leveraging the AIMC tiles, ultimately demonstrating up to 20.5x/20.8x performance/energy gains with respect to a SIMD-enabled fully-digital reference implementation.

In the next section, I look at how the tightly-coupled AIMC tile models can be easily refactored to model an alternative novel AI solution, and the subsequent exploration that followed.
3.4 Tightly-Coupled Systolic Arrays for Transformers

3.4.1 Systolic Arrays for Transformers

Figure 3.16: A high-level overview of a systolic array performing a matrix-matrix multiplication operation, recreated from Figure 2 left in [40]. The 3x4 yellow Input matrix is loaded into the systolic array, one element in each row every clock cycle in a staggered fashion. Weights are held stationary in the blue systolic array DPUs. The 4x3 green output matrix is obtained in a staggered format after less than 15 clock cycles.

Systolic arrays are highly parallel hardware architectures that use multiple data processing units (DPUs), typically arranged in a square matrix, to compute partial products on data that is output from adjacent DPUs, as shown in Figure 3.16. Input and weights matrices can be sent through the rows and columns, respectively, of a systolic array at each computation step to optimize matrix-matrix multiplication operations much faster than general-purpose processors. Each computation step also yields a diagonal row of the output. Furthermore, systolic arrays are highly scalable, with the primary bottleneck being very high area requirements \(O(n^2)\) as the systolic array grows in size. Therefore, the most attractive targets for systolic array-based acceleration are applications with matrix-matrix multiplication-based bottlenecks.

Transformers are a new class of neural network that are very popular with natural language processing, and more recently large language model-type, applications [1]. Transformers commonly consist of an embedding layer for inputs and encoder transformer blocks for processing data. Encoders are a common bottleneck in Transformer applications due to
multi-head attention nodes. These nodes have three weights matrices: $W^Q$, $W^K$, and $W^V$, which are multiplied by some input $X$ to obtain the query $Q$, key $K$, and value $V$ matrices, respectively. $Q$ and $K$ matrices are multiplied with scaling and softmax applied, followed by an additional multiplication operation with $V$.

![Figure 3.17: A 2x2 systolic array of weight-stationary DPUs, with the DPU model being recreated from Figure 2 right of [40]. The weights are programmed in each green cell before application run time. The inputs are propagated column-wise from a two element-tall matrix and an accumulator is propagated row-wise from a zero matrix during run time.](image)

Systolic arrays can be designed with weights stationary in DPUs, as shown in Figure 3.17, which optimizes data transfer and reuse as matrix-matrix multiplication operations propagate through the array. The weights matrices $W^Q$, $W^K$, and $W^V$ are only modified during training, so during inference they can be programmed into a systolic array DPUs.

### 3.4.2 Enabling TiC-SAT with ALPINE

Given the prevalence of multiplication operations in Transformers, they make for an attractive target for tightly-coupled systolic array-based acceleration. Ultimately, to use ALPINE for systolic arrays, the underlying generic PIO model was refactored to implement parameterizable DPUs with custom instruction programmed to load and unload the systolic array on a per-CPU core basis.

To implement TiC-SAT in ALPINE, the AIMC tile input, output, and weights arrays are repurposed for the systolic array matrix multiplication engine. This matrix multiplication engine is then interfaced with a tightly-coupled ISA extension with three new R-type instructions. First
is $SA_{LD}$, which loads four packed 8-bit values into DPUs at the TiC-SAT systolic array input in four clocked cycles. The next instruction is $SA_{IOC}$, which control input-output computation. This instruction activates the first-in first-out queues at the periphery of the systolic array and performs the MACs. The last new instruction is $SA_{IO}$ which perform input and output propagation of the array, but not the computation. This instruction is used for when the systolic array is not fully loaded.

### 3.4.3 TiC-SAT Dataflow

Given the large sizes of matrices in Transformers, TiC-SAT employs a matrix-matrix multiplication tiling technique to optimize small, partial matrix-matrix multiplication operations. This is already known to benefit even non-accelerated matrix-matrix multiplication operations due to the increased data locality [112].

The algorithm used TiC-SATs for matrix-matrix multiplication operations is as follows. First, the target matrices are split into tiles that can fit into the TiC-SAT systolic array (e.g., a 32x32 tile of the target matrices is extracted to use with a 32x32 systolic array). Next, $SA_{LD}$ is called to initialize the weights in the systolic array. Then in a for loop, the input matrix elements are streamed into the systolic array while the output matrix elements are streamed into the output matrix tile ($SA_{IO}$ and $SA_{IOC}$). This process was empirically optimized by using tile dimensions that make the size of the matrices slightly less than the L1 cache size of the system. Furthermore, the custom instructions, just as with AIMC tile-based ALPINE, can easily be implemented as intrinsics in C/C++ code.

### 3.4.4 TiC-SAT Experiments and Results

This implementation of tightly-coupled systolic arrays into ALPINE was then used to run the following exploration, which was performed primarily by the first author, Alireza Amirshahi. Here I report as a testament to ALPINE’s success the benefits of TiC-SAT using multiple BERT and VisionTransformer (ViT) applications [113], [114]. The simulated system is a small edge system running a single in-order CPU at 1GHz, with 32kB of L1 data and instruction caches, 1MB of L2 cache, and 4GB of DDR4 memory.

Using BERT-large as an initial case study, two baseline (reference) implementations were used to compare to the TiC-SAT-enabled system. The first is a non-optimized (N.O.) model with conventional matrix-matrix multiplication implemented as a thrice-nested for loop iterating on the input matrix as well as weight columns. The second is an optimized cache utilization (C-Opt.) model in which the input, weight, and output matrices are tiled into sub-matrices that fit in the L1 data cache. These reference implementations are then compared to TiC-SAT-enabled models with varying $k \times k$ sizes of systolic arrays. TiC-SAT was synthesized with the Synopsys Design Compiler and the TSMC 28nm library to estimate area. For fair comparisons to the state-of-the-art, the 28nm area estimations are then scaled down to 14nm. The total
area of a 16x16 TiC-SAT, including first-in first-out buffers for data alignment at the input and output, is estimated to be 0.044 mm$^2$. As a point of comparison, Gemmini and SMAUG take up an estimated area 0.858 mm$^2$ and 1.44 mm$^2$ with 22nm and 14nm FinFET technologies, respectively [30], [32]. In other words, the dedicated area for the loosely-coupled systolic array accelerators competing with TiC-SAT is up to 32.5x greater.

The scalability of the performance of varying TiC-SAT sizes was confirmed using BERT-medium before then comparing a variety of reference Transformers to those implemented with 16x16 TiC-SATs.

**BERT-large and BERT-medium Transformer Case Study**

<table>
<thead>
<tr>
<th>Model</th>
<th>N.O.</th>
<th>C-Opt.</th>
<th>4x4 SA</th>
<th>8x8 SA</th>
<th>16x16 SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-large</td>
<td>836.4s</td>
<td>144.2s</td>
<td>63.1s</td>
<td>22.6s</td>
<td>9.3s</td>
</tr>
</tbody>
</table>

Table 3.6: BERT-large Run Time Results

<table>
<thead>
<tr>
<th>Experiment</th>
<th>MHA</th>
<th>Projection</th>
<th>FF1</th>
<th>FF2</th>
<th>Non-GEMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>N.O. Baseline (Percent)</td>
<td>8.31%</td>
<td>3.88%</td>
<td>51.35%</td>
<td>36.42%</td>
<td>0.03%</td>
</tr>
<tr>
<td>N.O. Baseline (Real Time)</td>
<td>69.50s</td>
<td>32.45s</td>
<td>429.49s</td>
<td>304.62s</td>
<td>0.25s</td>
</tr>
<tr>
<td>16x16 TiC-SAT (Percent)</td>
<td>22.26%</td>
<td>7.13%</td>
<td>34.61%</td>
<td>32.92%</td>
<td>3.09%</td>
</tr>
<tr>
<td>16x16 TiC-SAT (Real Time)</td>
<td>2.07s</td>
<td>0.66s</td>
<td>3.22s</td>
<td>3.06s</td>
<td>0.29s</td>
</tr>
</tbody>
</table>

Table 3.7: BERT-large sub-ROI Run Time Results

Table 3.6 shows the total run times of BERT-large using the two different baselines (N.O. and C-Opt.) as well as three different TiC-SAT accelerator sizes. It highlights the impact of cache locality, by showing how cache optimization of a large Transformer can lead to a speedup of 5.8x. But even with that improvement, even a small 4x4 TiC-SAT gains over 2x speedup over the cache-optimized BERT-large. Scaling the TiC-SAT to 16x16 ultimately leads to a speedup factor of 89.5x.

Table 3.7 then shows the component (sub-region-of-interest or sub-ROI) run times for the N.O. and 16x16 TiC-SAT BERT-large experiments, where MHA refers to the multi-head attention layer, and FF1 and FF2 refer to the two feed-forward layers of the Transformer. All layers consist of GEMM operations other than the "non-GEMM" category which primarily includes activation and normalization functions.

The table shows that the GEMM operations of the feed forward layers are what primarily drive the performance of the Transformer, and therefore can be accelerated by TiC-SATs. Furthermore, even after acceleration, the MHA, Projection, and non-GEMM layers continue to be less significant for improving Transformer performance.
Further analysis shows that TiC-SAT not only speeds up Transformers by parallelizing operations, but by also reducing memory transaction overheads. Over four billion total read and write accesses (primarily done in the feed forward layers) in the N.O. experiment is reduced to only 4.5 million in the cache-optimized experiment, which is then further refined to under three million with the 16x16 TiC-SAT.

The reason for this reduction in read and writes is first, that a single SA_IOC instruction triggers \( k^2 \) MAC operations where \( k \) is the size of the TiC-SAT, thus lowering the number of required instructions, and second, \( k^2 \) parameters are resident during computation in the SA, thus reducing the size of the working set. \( k \) scales with the size of the TiC-SAT, and as shown, so too do the performance benefits.

### TiC-SAT General Results

<table>
<thead>
<tr>
<th>Model</th>
<th>( d_{seq} )</th>
<th>( d_{model} )</th>
<th>( h )</th>
<th>Parameters</th>
<th>16x16 TiC-SAT Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViT-base/16</td>
<td>197</td>
<td>768</td>
<td>12</td>
<td>86M</td>
<td>69.4x</td>
</tr>
<tr>
<td>ViT-base/32</td>
<td>50</td>
<td>768</td>
<td>12</td>
<td>86M</td>
<td>48.8x</td>
</tr>
<tr>
<td>ViT-large/16</td>
<td>197</td>
<td>1024</td>
<td>16</td>
<td>307M</td>
<td>82.5x</td>
</tr>
<tr>
<td>ViT-large/32</td>
<td>50</td>
<td>1024</td>
<td>16</td>
<td>307M</td>
<td>57.2x</td>
</tr>
<tr>
<td>ViT-huge/14</td>
<td>257</td>
<td>1280</td>
<td>16</td>
<td>632M</td>
<td>82.7x</td>
</tr>
<tr>
<td>BERT-tiny</td>
<td>512</td>
<td>128</td>
<td>2</td>
<td>4M</td>
<td>20.3x</td>
</tr>
<tr>
<td>BERT-mini</td>
<td>512</td>
<td>256</td>
<td>4</td>
<td>11M</td>
<td>38.2x</td>
</tr>
<tr>
<td>BERT-medium</td>
<td>512</td>
<td>512</td>
<td>8</td>
<td>41M</td>
<td>58.3x</td>
</tr>
<tr>
<td>BERT-base</td>
<td>512</td>
<td>768</td>
<td>12</td>
<td>110M</td>
<td>69.3x</td>
</tr>
<tr>
<td>BERT-large</td>
<td>512</td>
<td>1024</td>
<td>16</td>
<td>340M</td>
<td>89.5x</td>
</tr>
</tbody>
</table>

Table 3.8: Transformer Results

Table 3.8 is recreated from Table 2 of [40], and shows the parameters and speedups of all of the tested Transformer models, where \( d_{seq} \) is the model input sequence length, \( d_{model} \) is the vector length for each sequence, and \( h \) is the number of Transformer heads in the multi-head attention block. The combination of hardware acceleration and software optimization consistently leads to large speedups – up to 82.7x in the largest ViT model and 89.5x in the largest BERT model.

### 3.4.5 Tightly-Coupled Systolic Arrays: Key Takeaways and Conclusions

In this work, I showcased TiC-SAT, an ALPINE-based architecture and framework for modeling tightly-coupled systolic arrays targeting acceleration of transformer applications. I contributed to the model for systolic array acceleration in the validated ALPINE full system simulator, and defined its interface with a custom ARMv8 instruction set extension. TiC-SAT overcomes classical data transaction bottlenecks associated with other accelerated systems for transformers.
By targeting BERT and VisionTransformer models, we examined the speed-ups and memory behaviour relative to the reference and optimized applications. This exploration of transformer architectures and computer organizations has demonstrated substantial speed-ups, including up to 89.5x for the BERT-large transformer using a 16 by 16 TiC-SAT accelerator.

In the following section, I discuss how ALPINE can now be used to directly compare systolic array and analog in-memory computing-based solutions for edge CNNs using the ALPINE framework.

### 3.5 Cross-Solution Case Study: Comparing Caches, SIMD, Analog In-Memory Compute Tiles, and Systolic Arrays

#### 3.5.1 Competing Solutions for Machine Learning Applications

While surveys and field meta-analyses have been performed to compare the high-level attributes of different AI solutions, to this author's knowledge, no study has experimentally directly compared competing edge AI solutions at the edge for common applications at the full system level [115]–[119]. Indeed, a review of other prior art targeting AIMC and systolic arrays shows that baseline experimental comparisons, including of related works and the previously showcased works in Sections 3.3 and 3.4, always target comparisons to so-called "state-of-the-art" homogeneous systems, or systems that leverage a SIMD co-processor [29], [30], [32], [49], [76]–[87].

However, thanks to the introduction of the ALPINE framework, and the model validations and explorations in Chapters 3.3-3.4, it is now possible to compare the performance, energy, and application behaviour characteristics of multiple technologies side-by-side that leverage a common "tightly-coupled" interface [39], [120], thus achieving one of the goals set out by this thesis.

In this case study, I experimentally compare the performance, nuances, and energy of mapping and then performing inference on common CNN applications mapped to multiple systems, including those tightly-coupled AIMC tiles and TiC-SATs, as well as conventional "large cache" and SIMD-enabled systems. By leveraging ALPINE and gem5’s modularity and flexibility, I can not only compare these AI solutions to each other, but also to systems with SIMD co-processors and systems with very large caches.

#### 3.5.2 Overview of Competing Edge AI Solutions

In order to fairly compare tightly-coupled AIMC, TiC-SATs, SIMD co-processor, and more cache-enabled systems, we must consider the various constraints at play that would lead to the utilization of one technology over another for AI inference: namely, a trade-off in flexibility, energy, and area.
Chapter 3  Tightly-Coupled AI Accelerator Solutions

Fundamentally, general-purpose systems that leverage larger caches are the most flexible at the cost of energy and area. A conventional CPU will be able to provide integer, floating point, load, and store operations. Applications that suffer from memory-based bottlenecks will be able to leverage the large caches and memory locality to help minimize data transfers, which can be easily sized to the form factor of the overall system. However, as we know from previous sections, AI applications suffer a simultaneous compute and memory bottleneck, which motivates our heterogeneous systems.

The next most flexible solution we can consider are SIMD co-processor-enabled systems, such as those that use the ARM NEON SIMD unit. The SIMD co-processor brings in additional pipelines, registers, and highly parallel computation. They tend to require more energy than caches due to computation capability, but can represent balanced trade-off between compute and memory. These solutions are also not constrained to one kind of compute domain, and are considered state-of-the-art is numerous domains ranging from edge and internet-of-things domains, to high-performance computing and server-class computing domains [121]–[123].

However, AI solution architects that target edge domains want not only hardware that both simultaneously targets memory and compute bottlenecks, but hardware that can improve energy efficiency. This thesis chapter has already presented the two main software bottlenecks in AI inference: matrix-matrix multiplication in Transformers and matrix-vector multiplications in MLPs, RNNs, and CNNs. While we previously used systolic arrays for Transformers, multiple matrix-vector multiplication operations can be transformed or batched to turn them into matrix-matrix multiplication operations.

Therefore, we can consider the next most flexible solution to be systolic arrays. Weights must be stored prior to MMM processing, but using conventional SRAM cells, these weights are easily re-programmable. Therefore, systolic arrays are still flexible enough to be utilized for numerous AI-enabled applications while being specialized enough to often cost less area and energy than SIMD systems. The primary bottleneck encountered by systolic arrays is the area required for the memory cells in the data processing units [30], [32], [120].

AIMC tiles represent the most constrained solution because their weights must be programmed using costly (both in terms of compute and energy) line read-write operations prior to inference [124]. This is why numerous studies of AIMC tiles and associated technologies target edge AI applications that require retraining very few times and can run numerous inferences in between training and retraining cycles. However, through the use of PCM or resistive RAM cells, AIMC tiles can often fit more parameters in a given area than typical DRAM or SRAM technologies [125], [126]. As previously discussed in Chapter 3.3 as well, AIMC tiles also effectively remove weights from the working set of the edge AI-enabled application, which contributes to the resolution of memory-based bottlenecks [39].
3.5.3 Experimental Setup of Cross-Solution Study

AI Solution Area Estimations

As the available area for any of the aforementioned edge AI solutions chiefly determines the energy draw, cooling capability, compute power, and size of edge AI solutions, we use this metric as the basis for the experimental setup of a fair comparison of these technologies in full system level simulation. Prior art indicates that the ARM NEON SIMD co-processor fabricated using a 28nm process occupies roughly \(2 \text{mm}^2\) of area per CPU core, so we use this metric to size our caches and accelerators appropriately on a baseline system [23].

Table 3.9: System Configurations

<table>
<thead>
<tr>
<th>Component</th>
<th>More Cache</th>
<th>More Cache+</th>
<th>SIMD/TiC-SAT/AIMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>8 In-Order Cores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Frequency</td>
<td>2.3GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1D$</td>
<td>256kB</td>
<td>64kB</td>
<td></td>
</tr>
<tr>
<td>L1I$</td>
<td>256kB</td>
<td>64kB</td>
<td></td>
</tr>
<tr>
<td>L2$</td>
<td>16MB</td>
<td>32MB</td>
<td>1MB</td>
</tr>
<tr>
<td>RAM</td>
<td>8GB DDR4 @ 2.4GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disk</td>
<td>32GB Ubuntu 18.04 LTS w/ PCI Interface</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The chosen system parameters for our experiments are shown in Table 3.9. I consider a system with 64kB of private (per CPU core) L1 data and instruction caches, as well as 1MB of shared L2 cache, to act as the baseline system with which we apply accelerators or SIMD.

For the "more caches" system, we estimate the number of SRAM cells that could fit in an area of \(2 \text{mm}^2\) per CPU core. Given that the systolic array and AIMC tiles are fabricated using a 14nm process, we base our estimate off of 14nm fabricated 16kB SRAM subarrays in [127]. The exact amount of additional cache can vary widely due to differences in different kinds of fabrication processes (e.g. TMSC versus Intel), but a rough estimation indicates it is possible to obtain an additional 22-28MB of additional SRAM in our system [128]. Given that even server-class systems rarely exceed megabytes of L1 cache and our systems target the edge domain, we opt to increase the L1 cache sizes to 256kB and then test both a "lower end" and "higher end" more-caches system with 16MB and 32MB of L2 cache, respectively.

For the accelerator-enabled systems, we similarly use \(2 \text{mm}^2\) per CPU core as an objective and try to find the maximum dimensions that could theoretically fit in this area. For TiC-SAT accelerators, this means maximizing the number of systolic array SRAM cells, while for AIMC tiles, this means maximizing the size of the crossbar.

In the case of TiC-SATs, we scale the hardware model to 14nm from 28nm and fit the number of systolic array cells, as well as FIFO and input buffers (SRAM), to the constrained area [40]. Ultimately we obtain a maximum TiC-SAT size of 216x216.
Finally, in the case of AIMC tiles we consider the area to be the sum of the PCM crossbar, analog-to-digital (ADC) converter, and digital-to-analog (DAC) converter areas, based on [129]. The DAC array is sized according to the height of the crossbar while the ADC array is sized to the width of the crossbar. In general, DAC cell sizes are roughly a third the size of ADC cells, and so taller crossbars are favored over wider ones, since more PCM rows can match the DAC cells in the absence of DAC multiplexing. Ultimately, we estimate that one AIMC tile given 2mm² of area can obtain a crossbar size of 1224x256.

**AI Application Target**

![CNN Architecture Diagram](image)

**Figure 3.18**: (a) The architecture of the CNNs presented in [109] and their mapping onto the ALPINE systems. The blue boxes with the AIMC tiles represent the convolutional layers. The dense layers are not mapped to AIMC tiles. (b) shows the dimensions and parameters of each CNN. The CNN has five convolutional layers (three with Max Pooling), three dense layers, and ReLU activation functions for all layers except the last layer, which uses Softmax.

<table>
<thead>
<tr>
<th>Layer</th>
<th>CNN-F</th>
<th>CNN-M</th>
<th>CNN-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>224x224x3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conv1</td>
<td>64 11x11 kernels, stride 4, pad 0, x2 pool, LRN</td>
<td>96 7x7 kernels, stride 2, pad 0, x2 pool, LRN</td>
<td>96 7x7 kernels, stride 2, pad 0, x3 pool, LRN</td>
</tr>
<tr>
<td>conv2</td>
<td>256 5x5 kernels, stride 1, pad 1, x2 pool, LRN</td>
<td>256 5x5 kernels, stride 1, pad 1, x2 pool, LRN</td>
<td>256 5x5 kernels, stride 1, pad 1, x2 pool</td>
</tr>
<tr>
<td>conv3+4</td>
<td>256 3x3 kernels, stride 1, pad 1, x2 pool, LRN</td>
<td>512 3x3 kernels, stride 1, pad 1</td>
<td>512 3x3 kernels, stride 1, pad 1</td>
</tr>
<tr>
<td>conv5</td>
<td>256 3x3 kernels, stride 1, pad 1, x2 pool, LRN</td>
<td>512 3x3 kernels, stride 1, pad 1, x2 pool</td>
<td>512 3x3 kernels, stride 1, pad 1, x3 pool</td>
</tr>
<tr>
<td>dense1+2</td>
<td>4096 dropout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dense3</td>
<td>1000 dropout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AIMC params</td>
<td>1.7M</td>
<td>5.6M</td>
<td>5.5M</td>
</tr>
</tbody>
</table>

For a common target application to apply all AI solutions to, we use the Chatfield VGG8-variant CNNs [109], initially presented in Section 3.3.7. For convenience, the network model and parameters are copied here from the aforementioned section in Figure 3.18 (a) and (b), respectively.
In our test application we employ numerous software optimizations on all systems. This includes using pthreads to make the application multi-threaded, assigning one layer (with pooling and activation functions) to each CPU core in order to minimize data movement-based bottlenecks via pthread affinity, layer pipe-lining with ping-pong buffering to avoid I/O blocking, and an Eigen C++ back-end using tinystencils for optimizing data access and storage [66]. In our simulated system we run each CNN (fast, medium, and slow) for three inferences with each AI solution.

All CNN variants for each system are compiled with g++9, third-level optimization, and C++17. In the SIMD-enabled system, the "-O3" compiler flag automatically enables ARM NEON vectorization. In the more-caches, Aimc, and Tic-Sat systems, adding the "-DEIGEN_DONT_VECTORIZE" flag disables vectorization while maintaining other third-level optimizations.

TiC-SAT CNN Application Mappings

For the systolic array version of the application, we generalize the im2col transformation of the CNN convolutional layers to be a matrix-matrix multiplication of the im2col weights matrix of the convolution (comprising of all of the flattened kernels) and all flattened input patches.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rows x Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>363x64, 147x96, 147x96</td>
</tr>
<tr>
<td>conv2</td>
<td>1600x256, 2400x256, 2400x256</td>
</tr>
<tr>
<td>conv3</td>
<td>2304x256, 2304x512, 2304x512</td>
</tr>
<tr>
<td>conv4</td>
<td>2304x256, 4608x512, 4608x512</td>
</tr>
<tr>
<td>conv5</td>
<td>2304x256, 4608x512, 4608x512</td>
</tr>
<tr>
<td>dense1</td>
<td>9216x4096, 18432x4096, 12800x4096</td>
</tr>
<tr>
<td>dense2</td>
<td>4096x4096, 4096x4096, 4096x4096</td>
</tr>
<tr>
<td>dense3</td>
<td>4096x1000, 4096x1000, 4096x1000</td>
</tr>
</tbody>
</table>

An im2col transformation turns a convolutional operation into a matrix-vector multiplication (MVM) operation by first flattening kernels, and then arranging all kernels column-wise into a matrix. The dimensions of the im2col transformations (as well as dense layer dimensions) can be seen in Table 3.10. The input vector is the flattened "patch" of the input window covered by the kernel. The output of the MVM is then a 1x1 pixel of the output. The MVM must be performed for each pixel of the output window of the convolution [130].

im2col can be generalized into a matrix-matrix multiplication (MMM) operation, and therefore made usable with systolic arrays by arranging all input patches row-wise. The main drawback of this approach is that larger kernels shift the input patch from row to row, meaning a lot of
extra data is copied. This limitation can be overcome with rearranging the input buffer and FIFOs of TiC-SATS so that data reuse is propagated, rather than relying on sending of the same input again, but this is the subject of future work. Furthermore, because the systolic array doesn't match the size of the input and weights matrices, partial MMMs are performed where partial results must be stored and combined, similar to the matrix tiling utilized in [120]. The systolic arrays are applied to all convolutional layers of the CNN.

In cases where a partial MMM cannot completely fit in the TiC-SAT (for example, because the size of the weights matrix and TiC-SAT do not divide perfectly), standard CPU operations are used in lieu of padding for leftover MMMs, due to systolic array overhead with low TiC-SAT utilization.

**AIMC Tile-Enabled CNN Application Mappings**

Finally, in the AIMC tile-enabled version of the CNNs, an im2col transformation of each convolution is applied, where as many of the trainable parameters of the convolutional layers are stored in the AIMC tiles as possible. Given that AIMC tiles are optimized for MVM operations, we keep the MVM operation and perform one for each output pixel.

However, given the AIMC tile size of 1224x256, this means several layers of the CNN can only be partially computed using AIMC tiles, especially with Chatfield medium and slow variants. The portion of the convolution (partial im2col MVM) that cannot be performed with the AIMC tiles must be performed conventionally using digital operations, similar to the more-caches systems, and then combined with the partial accelerated output.

<table>
<thead>
<tr>
<th>Layer</th>
<th>AIMCv1 Layer Coverage</th>
<th>AIMCv2 Layer Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chatfield-F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conv1</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>conv2</td>
<td>77%</td>
<td>51%</td>
</tr>
<tr>
<td>conv3</td>
<td>53%</td>
<td>27%</td>
</tr>
<tr>
<td>conv4</td>
<td>53%</td>
<td>13%</td>
</tr>
<tr>
<td>conv5</td>
<td>53%</td>
<td>13%</td>
</tr>
<tr>
<td>dense1</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>dense2</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>dense3</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

The size of the weights matrices for all layers in the Chatfield CNNs using im2col transformations, as well as AIMC tile coverage of the layer (by percentage) is listed in Table 3.11. I do not apply the AIMC tiles to dense layers, as the low coverage and data locality would not yield any speedups. This observation, along with the fact that the input layer has a very small im2col transformation (363x64 in Chatfield-F and 147x96 in Chatfield-M and Chatfield-S), leads us to
propose two AIMC tile-enabled systems to evaluate.

The first AIMC tile-enabled system ("AIMCv1" in Table 3.11) is the baseline system with one AIMC tile per CPU core. In this configuration it is worth noting that the first CPU, which runs the input convolution, has 100% tile coverage, but only a small proportion (7.3%) of the AIMC tile crossbar is actually utilized. The second, third, fourth, and fifth convolutions all utilize 100% of the AIMC tile crossbar, which can end up covering only a small part of the total im2col size. The three dense layers utilize no AIMC tiles.

Therefore, between the three dense layers and the low AIMC tile utilization in the first convolutional layer, an eight CPU-core system with one AIMC tile per core is only using slightly more than half of its available accelerator resources. This motivates our second AIMC tile-enabled system ("AIMCv2" in Table 3.11), which assigns two 1224x256 AIMC tiles to four CPU cores which run convolutional layers conv2, conv3, conv4, and conv5. The result is that even though four CPU cores don't have AIMC tiles, 100% of the accelerator resources and area allocated to AIMC tiles is being used.

### 3.5.4 Cross-Solution Case Study Results and Analysis

![Figure 3.19: Matrix of results organized row-wise by metric (run time, energy, and L1 cache miss count per inference) for SIMD-enabled, AIMC-enabled, more-caches (16MB L2 and 32MB L2), and systolic array (TiC-SAT) systems. The results are arranged column-wise by Chatfield VGG8 variant (F(ast), M(edium), and S(low), for left, middle, and right columns, respectively). The stacked bars in the L1 cache miss graphs show the per-CPU cache misses. Note that the TiC-SAT energy does not include the accelerator energy, as there was no power model available in [120].](image)

The time, energy, and L1 cache miss results for all tested systems and CNNs can be seen in Figure 3.19. L1 cache misses per inference is a measure of data locality, in that more misses require going to shared cache or memory to access data. As shown in the results figure,
the data locality in different system configurations is very highly associated with the actual performance of the CNNs. Also of note is that the energy and speedup results are generally very similar, as the total energy spent on inference depends on the run time, even if the power varies in different systems.

To summarize the results, the "more-cache" systems perform similarly, with less than 1% deviation in performance statistics (time and energy) relative to one another. Using the "more-cache" systems as a baseline, the SIMD-enabled system achieves an average 3.7x performance improvement, while the TiC-SAT systolic array system slightly under-performs by achieving an average 3.4x performance improvement. The AIMCv1 system with low accelerator utilization attains an average 4.6x performance improvement, while the fastest and most energy-efficient system was the AIMCv2 system with 100% AIMC tile utilization at an average 6.3x performance improvement. A maximum performance improvement of 9.4x was achieved by the AIMCv2 system running Chatfield-F.

Performance of "More-Caches" and SIMD-Enabled Systems

In general, the reason why the more-caches systems perform similar is two-fold. First, the very large dense layers regularly thrash the shared last level cache, on account of their combined working set size (size of the weights, inputs, and outputs combined per inference) of 58.6MB, 96.4MB, and 73.3MB for Chatfield-F, Chatfield-M, and Chatfield-S, respectively. This reduces any benefits additional caches may have had, as the CPUs running the dense layers in the more-caches systems still end up waiting for memory up to eight times more than in other systems. Second, adding more cache only addresses data locality and not computation bottlenecks, which are simultaneously present. If the chosen neural network had a much smaller working set, adding more caches may have yielded more benefit.

It is for these reasons that the SIMD and accelerated systems are able to provide substantial performance gains, despite smaller caches.

In the case of the SIMD system, the combination of SIMD load/store instructions as well as vectorized multiplication and addition instructions yields a very significant speedup. Effectively, more data can be requested and computed upon more efficiently, meaning the compute and memory bottlenecks of the neural network are addressed simultaneously, and for all layers of the network (include dense layers).

Performance and Analysis of Accelerated Systems

The primary disadvantages of the accelerated systems as compared to the SIMD-enabled system (other than the loss of general purpose use) is that the accelerated systems don't have SIMD load and store instructions, and they are only applied to the convolutional layers.

In particular, the TiC-SAT system was generally 5% slower and less energy efficient than
the SIMD system. This result may be surprising, as we know that performance of TiC-SAT systems scale linearly with $k \times k$ systolic arrays. However, unlike the Transformers where TiC-SAT was applied in Chapter 3.4, multiple MMM operations are not combined and pipelined in the convolutional operation of the Chatfield CNNs. The partial MMMs are performed, and then sent to the next CPU core for further network operation. Furthermore, the lack of data reuse from the static TiC-SAT input mechanism requires copying and forwarding the same data multiple times, leading to poor locality. Therefore TiC-SAT systems have less opportunity to attain speedups for convolution operations. However, future work on TiC-SATs could incorporate configurable multi-directional array architectures to both reuse data more efficiently and handle computation of partial MMMs that don’t fit perfectly in the systolic array [131].

Despite the lack of SIMD load/store unit and application only to convolutional layers however, both the AIMCv1 and AIMCv2 systems both out-performed all other systems. AIMCv1 and AIMCv2 systems provided an average additional 25% and 72% performance improvement over the SIMD system, respectively. This shows that speedup and energy improvement in CNNs is not only a matter of how much acceleration resources are on the system, but how they are utilized. An additional average 52% performance improvement was achieved simply by rearranging the AIMC tiles and readjusting the mappings. How and why AIMC tiles should be mapped to particular layers is explored in more detail in Chapter 4.

### 3.5.5 Cross-Solution Case Study Takeaways and Conclusions

In this cross-solution case study, I performed the first comparison of multiple tightly-coupled accelerators targeting the same edge AI application to expose the strengths and weaknesses of competing edge AI solutions. By comparing systems that have more caches to SIMD-enabled systems, as well as tightly-coupled AIMC tile-enabled and TiC-SAT-enabled solutions, we show the importance of simultaneously addressing memory and compute bottlenecks, as well as how to extrapolate performance and energy benefits even with resources constrained by area in an oversized network. I ultimately achieved an average 3.7x speedup and energy improvement in neural network inference with a SIMD-enabled system over systems that had an addition 17.5-33.5MB of L1 and L2 cache. While TiC-SATs did not improve on the performance of the SIMD-enabled system, I highlighted strategies for overcoming systolic array-based bottlenecks in convolutions. I then achieved an additional average 25% and 72% performance improvement over SIMD systems using two different configurations of AIMC tiles.

### 3.6 Summary

In this chapter, I presented two competing tightly-coupled AI accelerator solutions and then a case study comparing them, thus achieving one dimension of heterogeneous architecture explorations set out by this thesis.
I started by using the ALPINE framework to implement analog in-memory compute (AIMC) tiles. I presented the AIMC tile workload data flow using an ARMv8 ISA extension and implemented a software library, AIMClib, to facilitate quick architectural exploration of AIMC tile-enabled systems. Then, I performed three case studies on two simulated edge systems showing where and how to attain performance and energy benefits from AIMC tiles. Using MLPs, I show how AIMC tiles target acceleration of the primary bottleneck of neural networks – MVMs – by reducing the sub-ROI time of the MVM from 96% of the run time to under 1%. I then showed how AIMC tile and cross-CPU core communication emerges as new bottlenecks, especially in MPSoCs. Using LSTMs, I then show how to parallelize and pipeline AIMC tile operation to speed up neural networks with time dependencies and numerous digital operations, further stressing cross-core communication bottlenecks. These studies were augmented with computational complexity and memory complexity analyses. Finally, using VGG8 variant CNNs, I showed how AIMC tiles could achieve over 20x speedup and energy benefits in MPSoCs while demonstrating the need to consider load balancing in more advanced applications.

I then contributed to a work refactoring ALPINE AIMC tiles to implement a competing edge AI solution: tightly-coupled systolic arrays, or TiC-SATs. In this work, I contributed to the development of a revised data flow that targets MMMs for acceleration and use Transformers as our target neural network, as well as optimize L1 cache utilization by tiling MMMs. This exploration shows how with very little area, systolic arrays can accelerate large Transformer networks, with up to 89.5x speedup in BERT-large using only a 16x16 TiC-SAT, and even out-perform other systolic array-based solutions Smaug and Gemini.

Finally, thanks to the implementation of AIMC tiles and TiC-SATs in ALPINE, I am able to perform a first-of-its-kind case study of competing edge AI accelerator performance to conventional SIMD and cache-enhanced systems. In this case I consider chip area as a chief constraint for what accelerator and cache resources can made available to an edge AI system running VGG8 variant CNNs on an eight-core MPSoC. This case study presents the flexibility and trade-offs in different edge AI solutions, ultimately showing up to 9.6x speedup and energy improvement in AIMC tile-enabled systems over enhanced cache systems, with up to 72% performance improvement over SIMD systems as well despite limited resources.

In the next chapter, I dive more deeply into the trade-offs between accelerator resources and area by expanding ALPINE to model loosely-coupled systems, and compare differently-coupled systems in real applications directly.
4 Loosely-Coupled Accelerator Solutions

4.1 Introduction

As we've seen in Chapter 3, heterogeneous edge AI systems that utilize accelerators often see performance benefits proportional to the size of the accelerator or the coverage of the neural network that accelerator provides. But the question of how many accelerator resources, such as AIMC tiles or TiC-SATs presented in Chapter 3, can be placed on a system is often subject to the proximity to other resources in that system. In other words, the number of AIMC tiles or TiC-SATs (or, indeed, any kind of edge AI accelerator) available within a system architecture is subject to the configuration of those accelerators. Generally speaking, there are "loosely-coupled" or "tightly-coupled" architectures, the latter of which was covered in the previous chapter.

Tightly-coupled accelerators are close in both proximity and interface to the system CPUs. They are able to perform data transactions with very low overhead, and therefore with very high performance benefit, at the cost of highly valuable chip area close to the CPU (area that competes with caches, peripherals, and other modules). This constrains the total number of parameters that can be held in the AIMC-based accelerator, or the number of TiC-SAT cells. In contrast, accelerators that are loosely-coupled are typically centralized in a cluster, are placed further away from the CPU, and are usually interfaced via peripheral buses (e.g., PCIe). The distance means more accelerator resources can be placed on the system, including dedicated digital logic for accelerating activation functions and connections for layer pipelines, which is often necessary to limit the slowdown associated with the high cost of data movement to and from the accelerator [38]. This trade-off in performance, accelerator resources, interface, and proximity to the CPU is central to the question of the usability of accelerators for modern and upcoming edge AI systems and the NNs run on them, and is therefore a dimension of heterogeneity that warrants architectural exploration.

However, a review of the literature reveals scant resources focusing on these trade-offs. Many works involving AIMC tiles (or similar in-memory compute accelerators) target NNs that are fully integrated within the accelerator. In other words, the underlying NN layers that are
typically targeted for acceleration (dense or fully connected layers, and convolutional layers) are usually assumed to fit completely in the accelerator, either by sizing the tile appropriately or limiting the size of the overall network \([132]–[134]\).

Fundamentally, this is counter to the general trend of NNs getting larger, deeper, and more varied over time. Therefore, it is infeasible to design, validate, and fabricate edge accelerators and systems that only target one particular workload or neural network. And so a question that remains unexplored is the behaviour of NNs when these accelerators are constrained and can only hold a finite number of parameters relative to the NN they are being used for, as well as when systems still benefit from accelerator in the face of such constraints \([135]\). Thus, ALPINE is necessary to facilitate explorations of how AIMC-based accelerators can be used with only partial coverage of neural networks.

In the following chapter, I enhance ALPINE by implementing loosely-coupled AIMC clusters and focus on AIMC as the primary accelerator to explore differently coupled systems, using modern edge CNNs as the primary neural network test case to explore methods of load balancing accelerator resources in edge MPSoCs. My contributions are as follows:

- I perform a study of hybrid accelerated and non-accelerated dense and convolutional layers with tightly-coupled AIMC tiles to show the relationship between accelerator coverage of NN parameters, NN compute intensity, and speedups.

- I propose a metric, "multiply-and-accumulate operations (MACs) per transfer rate", to capture the potential benefit of assigning AIMC tile resources to a particular NN layer.

- Using this metric and total MACs, I explore multiple tightly and loosely-coupled system architectures and how to simultaneously load-balance AIMC tiles and CPU workloads in multi-processor systems-on-chip (MPSoCs).

- I extend the ALPINE framework, which targets tightly-coupled AIMC tiles, to also model loosely-coupled AIMC clusters.

- I perform a detailed inference performance and energy analysis of five modern edge CNNs that shows the trade-offs in AIMC tile proximity, total AIMC tile resources, and accelerator proximity, culminating into a maximum 5.9x speedup and 5.6x energy gains for different CNNs, even with only partial (42%) AIMC tile coverage of parameters.

### 4.2 Related Work

#### 4.2.1 Coupling of AIMC Tiles

"Coupling" of accelerators in a system refers to both the proximity and interface of said resources. A summary of tight versus loose coupling can be seen in Table 4.1. With respect to AIMC tiles, what is usually explored in the literature are so-called "loosely-coupled" systems.
Table 4.1: Tight versus Loose Coupling of Accelerators

<table>
<thead>
<tr>
<th></th>
<th>Tight Coupling</th>
<th>Loose Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proximity to CPU</td>
<td>On-Chip</td>
<td>Off-Chip</td>
</tr>
<tr>
<td>Interface</td>
<td>ISA Extension, Direct Line</td>
<td>General Bus</td>
</tr>
<tr>
<td>AIMC Tile Capacity</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>Dedicated Digital Logic</td>
<td>Provided by CPU</td>
<td>Off-Chip</td>
</tr>
<tr>
<td>Data Transfer Bandwidth</td>
<td>$\propto f_{CPU}$</td>
<td>1–100GBps</td>
</tr>
</tbody>
</table>

These systems place their AIMC tiles further away from the CPU and use a bus-based interface, which incurs a much higher data transaction cost via DMA transfer over a bus, but comes with the benefit of having more area to expand accelerator capability, usually in the form of larger and more numerous crossbars that form a cluster. An AIMC cluster far away from the CPU can also have dedicated hardware logic for gate combination functions, activation functions, and more, depending on the actual target application set [38], [92], [133].

Relative to a loosely-coupled AIMC cluster, an AIMC tile that is tightly-coupled with a CPU has a very low data transaction overhead, such as ISA extension or MMIO. Furthermore, dedicated digital logic for activation functions or layer pipe-lining are unnecessary because they can be provided by the local CPU. However, a large accelerator fabricated close the the CPU competes for area with other components that could enhance CPU performance, such as larger or more numerous functional units, a SIMD co-processor, a more refined or expanded CPU pipeline, and so on. This contention for chip area potentially limits the size and capability of the accelerator itself [39].

To the authors’ knowledge there has been no direct comparison of these accelerator interfaces for heterogeneous systems, let alone AIMC tiles. It can likely be attributed to the lack of an architectural simulator capable of modeling both tight and loose coupling. However, thanks to the introduction of the ALPINE framework [39], I can model both tightly-coupled AIMC tiles and loosely-coupled AIMC clusters by expanding the peripheral input-output (PIO) device model for which ALPINE is based, as done in Section 4.4.

It is worth noting that the paradigms considered here require heavy CPU involvement. In other domains, CPU orchestration of accelerators is common to separate CPU compute and accelerator compute. Coherent buses such as CXL can assist in the independence of accelerators in general-purpose systems by granting direct memory access to the general memory hierarchy without CPU intervention, although it is common for the CPU to still handle operating system services such as page faults and exceptions [136], [137]. This is a topic for future work that is elaborated upon in Chapter 5.2.2.
4.2.2 Load Balancing Hardware Resources in Edge-AI Inference

Most systems, including edge systems designed for AI applications, are multi-processor systems-on-chip (MPSoCs) that include multiple CPU cores, shared caches, and shared memory. Therefore, the process of maximizing the performance benefit of accelerator resources is not only a matter of using the available resources, but balancing their use across the system. Failure to do so can lead to layer bottlenecks and resource under-utilization, i.e., numerous compute resources are waiting for one particular resource in order to proceed with computation.

Neural network frameworks such as PyTorch and tensorflow typically perform inference as a sequential per-NN process. Rather than parallelizing the run time of a single NN, these frameworks use a single process to run the NN and try to instantiate numerous processes for when multiple inferences need to be run. This leads to an inefficient use of memory resources by creating numerous copies of the NN weights, which can clog and thrash the memory hierarchy, leading to a general slowdown of the NN inference. This is a concern for neural network usability in real-time applications with hard deadlines, especially in safety-critical systems [138].

The bottlenecks created by this high degree of data distribution and movement are what motivated Wu et al. [139] to propose a granular pipeline-based layer scheduler to perform NN inference. In their proposal, the pipeline is first configured into a certain number of stages, and then available CPU cores are assigned to each stage. The goal of the scheduler is to balance the execution time of stages by first balancing the feature map data movement across stages and then iteratively bi-partitioning the network graph. It validates its proposal using three MobileNet-classed CNNs on a heterogeneous ARM big.LITTLE system, ultimately achieving up to 73% throughput improvement. Dagli et al. [140] expand on this work by prioritizing energy constraints in neural networks running on the Nvidia Xavier AGX SoC, but AIMC tiles fundamentally have fewer memory transactions and a different data flow in comparison to GPU-centric data flows. Nonetheless, [139] acts as the starting point for our load balancing approach presented in Section 4.5.

4.3 Mapping ML Kernels into Hybrid Analog-Digital Computing Layers

In order to better understand the effects and potential drawbacks of utilizing AIMC tiles that may not entirely cover NNs, I first look at the performance behaviour of individual dense/fully-connected and convolutional layers that are only partially accelerated. We refer to the accelerated, on-AIMC tile, portion of the MVM as "analog" and the non-accelerated, CPU-only portion of the MVM as "digital".
4.3.1 Hybrid Analog-Digital MVMs

Figure 4.1: The ways in which layers too large for an AIMC tile can be mapped and how subsequent computation is divided for the equation, \( Output = A \ast X \), where \( A \) is the matrix of weights and \( X \) is the input vector. The portion of the weights matrix \( A \) that can be placed in the analog (ANA) are shown using the crossbar figure (light blue boxes). (a) shows the case where all weights can fit in the accelerator. (b) and (c) show cases where the accelerator can cover the width and height of the weights matrix, respectively, but not both. (d) shows a piece-wise case where the accelerator can cover neither the width nor height of the weights matrix.

Consider that I want to perform the MVM computation of \( Output = A \ast X \), where \( A \) is the matrix of trainable weights with \( m \) rows and \( n \) columns and \( X \) is the input vector, of size \( m \). The output is also a vector of size \( n \). In general, there are four cases of AIMC tile mappings for a constrained layer to be considered, as shown in Figure 4.1.

The first case (equation 4.1) is the fully analog, fully accelerated case: all of the weights of \( A \) can be stored in the accelerator (shown with \( A_{ANA} \)), and so the entire MVM is accelerated. To perform an MVM, the input \( X \) is stored into the accelerator input memory, the MVM in the accelerator is spawned, and then the output is sent from the accelerator back to the CPU or memory.

\[
Output_{(a)} = A_{ANA} \ast X \tag{4.1}
\]

In the second case (equation 4.2), the accelerator can only fit enough weights from the weights matrix width-wise, but not height-wise. This means I must split the weights matrix into an analog portion \( (A_{ANA}) \) that is stored in the accelerator and a digital portion \( (A_D) \) that resides in the normal memory hierarchy. The input is split into two parts \( (X_1 \) and \( X_2 \)), where each part corresponds to the rows of the weight matrices stored in the accelerator and in memory. The complete MVM of the inputs and weights means performing two separate MVMs: first, the analog accelerated MVM of the partial input with the analog weights \( (Out_1 = A_{ANA} \ast X_1) \),
and second the digital "conventional MVM" of the other partial input with the digital weights ($Out_2 = A_D \ast X_2$). Once both computations are complete, the final result is the summation of the analog and digital MVMs ($Out_1 + Out_2$).

$$Output_{(b)} = Out_1 + Out_2 = A_{ANA} \ast X_1 + A_D \ast X_2$$  \hspace{1cm} (4.2)

The third case (equation 4.3) is similar to the second case, except the accelerator covers the height of the weights matrix instead of the width. Instead of splitting the input for the analog and digital MVMs, the input is reused in its entirety to perform both analog and digital MVMs ($Out_1 = A_{ANA} \ast X$ and $Out_2 = A_D \ast X$, respectively). Once both MVMs are complete, the results are concatenated ($[Out_1, Out_2]$).

$$Output_{(c)} = [Out_1, Out_2] = [A_{ANA} \ast X, A_D \ast X]$$  \hspace{1cm} (4.3)

In the last case (equation 4.4), the accelerator can cover neither the height nor width of the whole weights matrix, and so three MVMs must be performed, akin to a combination of the previous two cases. The accelerated MVM ($Out_{2A}$) result must be summed with the digital MVM of the weights matrix corresponding to the columns covered by the accelerator ($Out_{2D}$). This result ($Out_2$) is then concatenated with other digital MVM ($Out_1$) result.

$$Output_{(d)} = [Out_1, Out_2]$$
$$= [A_{D1} \ast X, (Out_{2A} + Out_{2D})]$$
$$= [A_{D1} \ast X, (A_{ANA} \ast X_1 + A_{D2} \ast X_2)]$$  \hspace{1cm} (4.4)

Note that this analysis assumes no accuracy degradation. The ADCs need to have a higher precision than the PCM crossbar cells in order to house the result of the MACs. For hybrid analog-digital computation, I assume partial result summation and concatenation is done purely in digital (by the CPU). In our experiments, I assume the whole network has been quantized to 8 bits. The field in general is moving towards low-precision (3-4 bit) implementations [141].

### 4.3.2 Hybrid Layer Study Experimental Setup

In order to quantify the benefits of applying limited AIMC tiles to large NN layers and motivate our architectural exploration, I perform two studies on typical speedup targets for AIMC tiles: dense layers and convolutional layers. The dense layers perform single MVMs while
Loosely-Coupled Accelerator Solutions Chapter 4

Convolutional layers reuse weights and inputs to further optimize MVM iterations. In both studies, I use ALPINE to simulate a single-core system with a variable amount of tightly-coupled AIMC tiles in full system (FS) mode, using the full software stack including Linux kernel and file system.

The system specifications for all of these experiments are listed in Table 4.2. We consider 8-bit NN dense and convolutional layers with variable amount of 256x256 tiles in different mappings based on [142].

In multiple-tile mappings of NN layers, I share the input and output memories in the same model while assuming the same 100ns per MVM. For example, if given two AIMC tiles, I experiment with a 512x256 AIMC tile configuration, where this configuration has an input and output memory size of 512 and 256, respectively, as well as an experiment of the inverse configuration, that has an input and output memory size of 256 and 512, respectively, with a 256x512 crossbar.

<table>
<thead>
<tr>
<th>ISA</th>
<th>ARMv8 w/ AIMC Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>In-Order @ 2.3GHz</td>
</tr>
<tr>
<td>L1D/I Cache</td>
<td>64kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1MB</td>
</tr>
<tr>
<td>RAM</td>
<td>DDR4 @ 2400MHz</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 5.4</td>
</tr>
<tr>
<td>Filesystem</td>
<td>Ubuntu 16.04 LTS</td>
</tr>
</tbody>
</table>

The dense and convolutional layers are implemented using Eigen C++ library tensors to optimally facilitate data storage during inference. The Eigen C++ library is the backbone of tensorflow, and it offers highly optimized data storage [66]. We run 10 iterations of the layer for each experiment and report the average (equivalent to 10 inferences if each layer is a single NN). Run time statistics are gathered by isolating the region of interest (the 10 inferences) using gem5’s m5 binary and then reviewing the generated gem5 statistics file.

4.3.3 Dense Layer Study and Results

For the dense layer study, I consider three square dense layers, where the weights matrix is 1024x1024 (dense1k), 2048x2048 (dense2k), or 4096x4096 (dense4k). We then vary the number of tiles applied to the layer, ranging from zero 256x256 tiles (a fully digital, non-accelerated baseline case) to 256 tiles total, fully covering the 4k dense layer. It is assumed that the AIMC tile controller can orchestrate all of the tiles in the system, i.e. only a single input memory is needed to beam inputs across all rows of all tiles that are aligned. Likewise, this assumption is applied to multiple tiles with connected columns performing analog MACs that are sent to a shared output memory.
The speedup versus AIMC tile coverage results can be seen in Figure 4.2. With 100% AIMC tile coverage (i.e., there are enough AIMC tiles to hold all weights for the layer), the upper bound on speedups are 2.3x, 29.4x, and 106.2x for the 1k, 2k, and 4k dense layers, respectively. The increasing speedup with layer size is the result of the AIMC tiles decreasing compute complexity and removing weights from the working set (i.e., the memory hierarchy never sees the analog weights during inference).

Although the potential for speedup increases as the dense layer grows in size, it does so at the cost of needing more and more AIMC tiles. Only four 256x256 tiles are necessary to cover the entirety of the 1k dense layer, but 64 tiles of the same size are required to cover the entirety of the 2k dense layer and, likewise, 256 tiles to cover the entirety of the 4k dense layer.

It is likely impractical for a system to be fabricated with 256 tightly-coupled AIMC tiles per CPU core, so it is worth noting the inflection point present in Figure 4.2: while all dense layers converge to different magnitudes of speedup, all layers only start to realize speedups once more than 50% of the dense layer is covered by AIMC tiles. This is partially due to the necessary data transaction overhead – in general, CPU functional units are faster than transferring data to and from the AIMC tile and its 100ns MVM cost, if the data set is small enough. Therefore, in the context of larger NNs, it doesn’t make sense to apply AIMC tiles to dense layers without a large number of AIMC tile resources being allocated.

Another interesting note from the dense layer study comes from an analysis of the number of multiply-and-accumulate operations (MACs) in each of the dense layers. In general, the
The number of MACs in the dense layer is equal to the size of the vector input multiplied by the number of weights ($M A C s = A \times X$). The number of MACs in a NN is commonly used as a metric of how large and computationally intense a NN is [143].

In this chapter, I make a distinction between total MACs, digital MACs (DIG MACs), and analog MACs (ANA MACs). DIG MACs are the MACs that are non-accelerated and performed only by the CPU. ANA MACs are those performed by AIMC tiles. To create Figure 4.3, I calculate the total MACs for the layer, and then subtract that number of ANA MACs from the total MACs for each experiment (one data point representing a different percentage of AIMC tile coverage) to get the leftover DIG MACs.

The figure reveals a linear relationship between the DIG MACs and the run time, meaning that for dense layers, the MACs per second rate can be easily predicted as a function of the tile coverage, i.e., a system's rate of doing MACs is preserved in dense layers regardless of AIMC tile coverage.

4.3.4 Convolutional Layer Study and Results

For the convolutional layer study, I select convolutional layers of different types in order to gain insight into the relationship between DIG MACs count, speedups, and AIMC tile
coverage of large convolutional layers. This is because unlike dense layers, which only have two parameters controlling size and compute intensity (input and output sizes), convolutional layers have numerous parameters that contribute to both the size of the weights matrix and the frequency of its use, including number of output pixels, kernel size, number of input channels, and number of filters.

To pick several convolutional layer types to study, I first plot the different sizes of different layers in modern CNNs – namely MobileNetV2, ResNet50v1.5, and VGG16 – as shown in Figure 4.4 [65], [144], [145]. We use the im2col transformation for analog convolutional operations as it is the most optimized for AIMC tile-enabled MVMs. Note that CPU-only or SIMD implementations also use an im2row transformation, depending on the data layout [146]. Based on this plot, it is clear that the im2col weights matrices of most convolutional layers are very tall, and so I pick a variety of tall im2col transforms, ranging from an input layer (which are usually less than 1000 parameters total) to large deep layers with over 4000 parameters of height, as well as one very wide layer. In this work I do not target depth-wise convolutions for acceleration due to their poor AIMC tile utilization [147]. The selected convolutional layer parameters are shown in Table 4.3.

Similar to the hybrid layer study of dense layers already presented, I also compare each of the aforementioned convolutional layers’ performance after applying a variable number of AIMC tiles that are 256x256 in size. In addition to considering AIMC tiles mapped height-wise first (i.e., stacking tiles on top of each other to prioritize covering whole columns), I also perform equivalent experiments considering AIMC tiles mapped width-wise first (i.e., stacking tiles next to each other to prioritize covering whole rows).
Table 4.3: Hybrid Layer Study: Selected Convolutional Layer Parameters

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input Size</th>
<th>Kernel Size</th>
<th>Filters</th>
<th>Output Dim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV2</td>
<td>7x7x1920</td>
<td>1x1</td>
<td>320</td>
<td>7x7</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>14x14x1024</td>
<td>1x1</td>
<td>256</td>
<td>14x14</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>7x7x512</td>
<td>3x3</td>
<td>512</td>
<td>7x7</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>7x7x512</td>
<td>1x1</td>
<td>2048</td>
<td>7x7</td>
</tr>
<tr>
<td>VGG16</td>
<td>8x8x256</td>
<td>3x3</td>
<td>256</td>
<td>8x8</td>
</tr>
<tr>
<td>VGG16</td>
<td>8x8x256</td>
<td>3x3</td>
<td>256</td>
<td>8x8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layer</th>
<th>im2col Size</th>
<th>Total MACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV2</td>
<td>1920x320</td>
<td>30.1M</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>1024x256</td>
<td>51.4M</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>4608x512</td>
<td>115.6M</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>512x2048</td>
<td>51.4M</td>
</tr>
<tr>
<td>VGG16</td>
<td>27x64</td>
<td>1.8M</td>
</tr>
<tr>
<td>VGG16</td>
<td>2304x256</td>
<td>37.7M</td>
</tr>
</tbody>
</table>

Figure 4.5: Convolutional layer study graph showing factor speedup over digital reference versus layer coverage by AIMC tiles. Each point represents one experiment where 10 inferences are run on a single layer with some percentage of AIMC tile coverage. The red dotted line indicates 1x speedup over digital reference (analog and digital run times are equal).
Figure 4.6: Convolutional layer study graph showing the number of DIG MACs versus the run time of the dense layer. The "total" DIG MACs are calculated by first computing the total DIG MACs, and then subtracting the number of MACs that are performed by the AIMC tiles, given a certain percentage of coverage. Each point represents one experiment where 10 inferences are run on a single layer with some percentage of AIMC tile coverage. In general, further right on the graph is less AIMC tile coverage of the layer. Points with for the same layer with the same run time but different total DIG MACs value show the difference between height-wise coverage and width-wise coverage by the AIMC tiles.

Analogous to the dense layer study, our speedup results can be seen in Figure 4.5. The upper bounds on speedup (given 100% layer coverage with AIMC tiles) are 2.1x for MobileNetV2's bn7_conv2, 2.3x, 17.4x, and 2.7x for ResNet50v1.5's residual convolutions 4c-2a, 5b-2b, and 5b-2c, respectively, and 2.0x and 9.2x for VGG16's conv1 and conv6, respectively. The largest speedups are associated with the tallest layers tested (ResNet50v1.5 res5bb2b and VGG16's conv6). It is worth noting that even though the maximum possible speedups are less than what was obtainable with dense layers, the number of AIMC tiles needed to attain full layer coverage is also significantly lower. The largest of the layers tested, ResNet50v1.5's res5bb2b, only needs 18 AIMC tiles to achieve its maximum 17.4x speedup, in comparison to the 4k dense layer's 256 tiles to achieve 106.2x speedup. All other convolutional layers require even less tiles.

Additionally, unlike with the dense layers, there doesn't seem to be a clear inflection point amongst convolutional layers as to when speedups start to occur based on AIMC tile coverage. The threshold for the various convolutional layers occurs between 55% for larger layers and 85% for smaller layers.
Comparing the run time versus the number of DIG MACs for each layer also shows differing trends from the dense layer study. As seen in Figure 4.6, while there is still a linear correlation between DIG MACs and run time for each layer, the slope of a fitted line for each convolutional layer seems to differ from layer to layer. Furthermore, it takes more time per DIG MAC if the AIMC tiles are placed width-wise first, rather than height-wise first, owing to the column-major layout of Eigen C++ Tensors.

The difference in speedup trends can be explained by the data transaction costs for each layer. The number of AIMC tile queues (sending of input data) and dequeues (retrieving of post-MVM output data) performed is both dependent upon the input and output dimensions of the AIMC tiles, as well as the size of the kernel in each convolutional layer. A larger kernel leads to data reuse due to overlapping input windows, leading to less queues and thus more potential speedup. This is outlined most succinctly if I heuristically calculate the analog MACs rate per data transfer, using Equation 4.5:

\[
\text{MACs}_{\text{ANA}} = \frac{\text{MACs}}{((Qs/K + DQs) \times Y)} \quad (4.5)
\]

\(\text{MACs}_{\text{ANA}}\) is the total number of MACs performed in the AIMC tile. Given complete tile coverage of the layer, the analog MACs are the same as the total MACs. Queues and Dequeues are the data transfers to and from the AIMC tiles’ input and output memories, which in the case of complete tile coverage, corresponds to the input and output dimensions of the im2col transformation of each layer, respectively. \(Y\) is the number of pixels of the output dimension of the convolution, and is the number of MVM iterations per inference. Finally, \(K\) is the number of pixels in the kernel window, which is divided due to input data reuse (e.g., from shifting the input memory). It should be noted that the stride of the convolutional layer could also be taken into account by further dividing \(K\), but empirically high strides are used only in input layers, which are already predicted to have a low \(\text{MACs per transfer}\) rate due to the limited sizes used for edge applications.

The correlation between MACs, data transfer, and speedup, is then easily apparent in Table 4.4. A MACs per transfer rate on the order of magnitude of 0-400 leads to roughly 2x speedup, while
increasing the MACs per data transfer rate to above 1000, as in the case of VGG16’s conv6 and ResNet50v1.5’s res5bb2b layers, leads to significantly higher speedups. It is also worth noting that even though ResNet50v1.5’s res5bb2c has a similar order of magnitude of parameters, the wide layout of the layer in tandem with the data reuse factor (kernel size) works against the column-major layout of Eigen Tensors.

4.3.5 Hybrid Layer Study Takeaways and Limitations

In summary, through this study of hybrid analog and digital dense and convolutional layers, I were able to explore the usability of limited AIMC tile resources, as applied to single layers.

The hybrid dense layer study shows that while high speedups are attainable for large layers, the number of AIMC tile resources required is also very large. Furthermore, due to data transaction costs, speedups are only attainable once more than 50% of a layer is covered. Therefore, in large NNs with a variety of layers, dense layers should not be the first targets for acceleration.

Meanwhile, the convolutional dense layer study shows that high speedups are attainable for large layers that have a high factor of data reuse via large kernels, at a fraction of the number of AIMC tiles. 2x speedup is generally attainable for layers with full analog coverage, but data reuse can exploit the number of MACs performed per data transfer, thus yielding much higher speedups: up to 17.4x on our tested set of convolutional layers. Furthermore, due to the column-major layout of Eigen C++ Tensors, it is more optimal to have AIMC tiles cover a layer column-wise first, rather than row-wise.

To be able to exploit these features in a larger NN however, I must consider that modern edge NNs are first, very large (on the order of tens of millions of parameters, and growing [65]) and can consist of tens or hundreds of layers, and second, typically run on multi-core systems. The conventional strategy to speedup NN computation is to leverage parallel CPUs and operate NN pipelines (i.e., multiple sequential layers run in parallel and synchronize with other layers). The hybrid layer study gives direction on which layers to target for acceleration but not on how to balance computation in large-system, large-NN scenarios. Thus, I examine and implement the loosely-coupled AIMC paradigm in Section 4.4, and then look at load balancing strategies in MPSoCs in Section 4.5.

4.4 ALPINE Loosely-Coupled Framework

4.4.1 Tightly Coupled AIMC Tile Model

To model tightly-coupled AIMC tiles, a generic peripheral IO (PIO) device was implemented with a connection directly to the gem5 ISA interface in [39]. Custom instructions were added to gem5 using unused op codes, that then generate classes that can query the PIO device directly. Timings are set by simply setting the operation latency of the custom instruction's functional unit. The AIMC tiles are generated by the PIO device, which acts as an arbiter to
query individual, per-CPU core AIMC tiles. The bandwidth of the CPU connection to the AIMC tile is therefore subject to the CPU core frequency. Given 32-bit ISA extension-based transfers at a clock speed of 2.3GHz, a maximum bandwidth of 9.2GBps per CPU core is achieved in our tightly-coupled systems.

One key software constraint imposed by tightly-coupled AIMC tiles is that when a layer is programmed to a specific AIMC tile associated with a specific CPU core, that layer is only accelerated when it is run on that CPU. Given this limitation, loosely-coupled AIMC clusters have two primary advantages. First, AIMC tile clusters act independently from different CPUs, and therefore CPUs can employ work-stealing as a means of efficient task delegation [148]. Second, these clusters can readily assign as many AIMC tiles as necessary (or available) to selected NN layers, regardless of which CPU operates on that layer. In order to enable architectural explorations of this flexibility trade-off, I implement support for loosely-coupled configurations of AIMC cluster-enabled systems.

### 4.4.2 Modeling and Implementation of Loosely-Coupled AIMC Tiles

The target system model I implement in this work can be seen in Figure 4.7. In general, peripheral devices in gem5 are interacted with via the m5 packet system, whereby packets of data are sent and received by various clocked components (e.g., buses, memory, other peripheral devices) at specific events and times in the general event queue. Every gem5 PIO device must implement a read and write method that returns a tick (response) delay and interacts with a received packet.

To implement loosely-coupled AIMC clusters, I leverage ALPINE’s generic PIO device to implement a bus-based interface [39]. We extend the generic PIO device to use the read and write packet interfaces to perform the same computation and data delegation tasks as performed by the ALPINE ISA extension, i.e., queueing input data, spawning MVMs, and dequeueing output data are all possible using the PIO device read and write interfaces. We keep the same 1ns delays for 32-bit data storage/retrieval from the AIMC tile as well as 100ns for the MVM. These timings can be made dynamic however, depending on the exact cluster configuration (e.g. if multiple layers are pipelined on-board the cluster). It is worth noting...
that these delays are independent of the IO bus’ data transfer overhead, which is handled by a separate bus model.

For faster and more dynamic simulations, certain addresses are designated as configuration space addresses where the AIMC cluster model can dynamically reconfigure itself during simulation run time, including resizing AIMC tiles and on-cluster activation functions. This dynamic resizing can emulate multiple AIMC tiles as one AIMC tile that has been resized to the equivalent number of rows and columns.

4.4.3 Loosely-Coupled AIMC Cluster Interface

In order to interface the loosely-coupled AIMC tile cluster, I reserve an address space for contiguous reads and writes. We interface the IO bus via /dev/mem. Using this interface in gem5, I were able to achieve an empirical maximum throughput on our test system of 52GBps using an 8-CPU core multi-threaded read-write microbenchmark, which is on the same order of magnitude as PCIe 5.0-6.0 with x16 channels [149].

At the application level, I expand the C++ header-only library AIMClib from [39] to overload queue, dequeue, and MVM operations for the loosely-coupled system. By using the compiler flag -DLOOSELY_COUPLED, AIMClib will now automatically use the loosely-coupled system-based methods for interacting with the AIMC cluster. This way, I can still maintain the usual AIMC tile data flow of first queueing input data to the AIMC cluster input memory, spawning the MVM, and then dequeueing output data from the AIMC cluster output memory.

4.4.4 Software Support: Expanding AIMClib with tinytensorlib

In order to test a wide variety of NNs on simulated MPSoCs in ALPINE with tightly- and loosely-coupled AIMC-enabled configurations, I built a minimalist inference-focused library that builds atop AIMClib to provide native support for AIMC tiles: tinytensorlib.

A diagram of tinytensorlib’s relationship to AIMClib is illustrated in Figure 4.8 (note that this is similar to Figure 2.4.2, but is specifically targeted toward usage with AIMC tiles). tinytensorlib is a header-only C++ library that builds atop AIMClib by providing Eigen C++ Tensors as the primary data structure for NN layers. It includes support for queueing and dequeueing vector tensors for dense layers mapped to AIMC tiles, as well as queueing patches and dequeueing pixels for convolutional layers mapped to AIMC tiles. Outside of AIMC tile support, it provides prototypes and connections for numerous kinds of layers with normalization and activation functions, as well as support for ping-pong buffering in multi-core systems to avoid IO blocking. Because Eigen C++ Tensors are the back-end, SIMD functionality can be enabled by simply compiling with the ".O3" flag in g++.

With tinytensorlib, I am able to rapidly prototype inference applications to examine the performance of MobileNets, ResNets, and other CNNs in ALPINE with ease. Using Python
Figure 4.8: A diagram of tinytensorlib’s placement and features relative to AIMClib and a NN application. tinytensorlib can inherit from AIMClib to incorporate tightly- and loosely-coupled AIMC interfaces, but can also exclude AIMClib for use with digital-only applications.

code generation scripts, I can even generate code for our load-balanced networks that can then be cross-compiled for our target simulated system.

4.5 Load Balancing Methodology

In the following subsections, I propose four load balancing algorithms in total. These algorithms comprise both a loosely and tightly-coupled variant of two load balancing objectives. The goal is to simultaneously balance AIMC resources and CPU workloads, with the exact implementation subject to the constraints of each form of coupling.

4.5.1 Load Balancing Approach and Objectives

Given a MPSoC that runs a specific NN, I propose parallelizing the execution of the NN by grouping sequential layers into “workloads” that are run on different CPUs. I limit cross-CPU core communication overheads by creating workloads only at a layer-wise granularity, which avoid partial computation and synchronization of partial results. Then to balance the computational load of the CPUs, I use the number of digital multiply-and-accumulate operations (DIG MACs) in the NN as a means of quantifying NN computational intensity, and divide it by the number of CPU cores.

The goal of efficiently using accelerator resources is to then allocate our AIMC tiles or clusters in a way that accelerates the layers most critical for system performance. There are numerous objectives that can be set for how to assign AIMC tiles to layers, but for this study I consider two different approaches, visualized in Figure 4.9 and described below.

**REF Load Balancing:** The first algorithm, “reference” (REF), simply aims to distribute AIMC tile resources evenly across a NN. I consider this algorithm state-of-the-art as it only relies on DIG MACs for balancing AIMC tile resources. The number of DIG MACs after AIMC tile
Figure 4.9: Figure showing the steps of the reference and proposed load balancing algorithms, "reference" (REF) in (a) and "Prioritize Largest Layer" (PLL) in (b). The graphs show a simple NN with two convolutional layers in yellow boxes in a series of graphs, where the Y-axis is the number of DIG MACs for each of the two layers. At each step of each algorithm, one AIMC tile (blue boxes) is assigned.

allocation can be calculated by subtracting the number of analog MACs done in the AIMC tile from the total MACs for the layer the AIMC tile is assigned to. The result is that given multiple layers within a workload with a similar, large number of MACs, the AIMC tiles will be as evenly distributed as possible across these layers.

Given our conclusions from the hybrid layer study, I assert that exponential speedups over digital reference are attainable with more AIMC tile layer coverage (maximizing the MACs per transfer rate), convolutional layers gain more speedup with less AIMC tiles, and there is a data movement overhead that may cancel out acceleration benefits. These insights would suggest that the reference load balancing algorithm presented above may distribute resources too much, incurring more data movement overheads that are a detriment to the speedups obtainable by AIMC tiles. Therefore I investigate an alternative load balancing algorithm: Prioritize Largest Layer (PLL).

PLL Load Balancing: The second algorithm, "prioritize largest (in terms of MACs) layer", derives from our conclusions to the hybrid layer study. Rather than distributing AIMC tile resources across multiple layers evenly, PLL will instead prioritize placing all parameters of the selected layer in AIMC tiles (making the DIG MACs zero) if the AIMC tile resources are available, before selecting another layer to assign AIMC tiles to.

Recall that our proposed metric of MACs per transfer (equation 4.5) showed the number of
MACs performed in constant time per some number of data transfers, where the number of data transfers represents the accelerator overhead. The objective then is to capitalize on a high MACs per transfer rate, and therefore attain the highest speedups and energy gains by first selecting the layer with the highest MACs per transfer rate and then assigning as many AIMC tiles to individual layers as possible. Of course, the trade-off here relative to the REF load balancing algorithm is that the accumulated smaller speedups across many layers may out-perform the larger speedup of only a single layer.

There are other trade-offs to these load-balancing approaches that are specifically dependent on the coupling of system resources, which is explained in the following subsections.

### 4.5.2 Load Balancing Tightly Coupled AIMC Tiles and CPUs

In general, AIMC tiles tightly-coupled to specific CPUs are orchestrated only by their local CPU, thus requiring a thread affinity for a workload in order to benefit from acceleration. In the context of a NN where layers are pipelined and a subset of layers use AIMC tiles, the benefit is that each CPU can be statically assigned a portion of the inference before run time, with a guarantee of data locality. The primary drawback however is that AIMC tiles are more distributed, and if one CPU wants more AIMC tiles to cover more of a high impact layer, it is impossible for another CPU to "reallocate" an underutilized AIMC tile.

```
threshold = Total DIG MACs in NN / Number of CPUs
Current CPU = 0
Current CPU DIG MACs = 0
For layer in NN:
    Assign layer to Current CPU
    Current CPU DIG MACs += layer DIG MACs
    if current CPU DIG MACs > threshold:
        Current CPU = Next CPU
        Current CPU DIG MACs = 0
For each CPU:
    For each unallocated AIMC tile:
        Assign tile to to layer with most DIG MACs being run by CPU
```

Figure 4.10: Pseudo-code for the reference (REF) load balancing approach for tightly-coupled systems. This is a "top-down" approach where first layers are assigned to CPUs, and then subject to the number of tightly-coupled AIMC tiles per CPU, tiles are assigned to layers.

Because NN layers being locally mapped to AIMC tiles dependent on specific CPUs for orchestration, I propose a "top-down" approach where layers are first sequentially allocated to CPUs based on a total MACs threshold, and then the AIMC tiles for each CPU core are allocated to layers being run by each CPU. Rough pseudo-code for this can be seen in Figure 4.10, where the REF load balancing approach is applied. To apply PLL, the only change to this algorithm is that when a layer with the highest DIG MACs per transfer rate is found, AIMC tiles are applied until AIMC tile resources are zero or the entire layer is covered.
4.5.3 Load Balancing Loosely Coupled AIMC Clusters and CPUs

In loosely-coupled AIMC cluster-enabled systems, the AIMC cluster is disjoint from each of the CPUs, so each CPU can independently access the cluster and request MVMs for different NN layers. This means loosely-coupled systems can enable a run time optimization known as work stealing. Work stealing is when, given multiple tasks to be run by any CPU, the CPU that is next available (has finished running its current task and is therefore idle) is able to request the next task to be run, rather than potentially being forced to wait for the previous task from another CPU to finish (which is the case in tight coupling). Given a large NN with a variety of layers with different compute and memory intensities, work stealing may be beneficial to ensure maximum throughput of compute resources, at the potential cost of more data transaction overhead (e.g., the sharing of inputs and outputs from each layer across cores) [148], [150]. I anticipate that this overhead is diminished as more AIMC tiles are utilized in the system because they effectively remove weights from the algorithm working set (the data requirements per inference).

While unallocated tiles > 0:
Assign tile to the layer with most DIG MACs
threshold = Total DIG MACs in NN / Number of CPUs
Current CPU = 0
Current CPU DIG MACs = 0
For layer in NN:
Assign layer to Current CPU
Current CPU DIG MACs += layer DIG MACs
if current CPU DIG MACs > threshold:
  Current CPU = Next CPU
  Current CPU DIG MACs = 0

Figure 4.11: Pseudo-code for a reference load balancing algorithm for loosely-coupled systems. This is a "bottom-up" approach where tiles are allocated to layers first, and then layers are assigned to CPUs. To assign a tile to the layer with most DIG MACs, I simply iterate over the layers of the NN and calculates the DIG MACs for each layer.

Because of work stealing, it is relatively straight-forward to load balance loosely-coupled systems. I use a heuristic "bottom-up" approach, whereby AIMC tiles are assigned to NN layers first, and then the remaining DIG MACs are balanced by assigning layers to workload groups that can be run by any CPU (without thread affinity). Rough pseudo-code for this can be seen in Figure 4.11, where the REF load balancing approach is applied. Like before, PLL can be applied by simply changing the AIMC allocation logic to only assign AIMC tiles to another layer when the selected layer has had all parameters placed in AIMC tiles or there are no more AIMC tiles available.
4.6 ALPINE Load Balancing Experimental Setup

4.6.1 ALPINE Target Systems

For our simulated systems, I target the same mid-range edge device simulated in Table 4.2, but I expand our simulated system to be a MPSoC with 8 CPU cores available.

We perform NN experiments on six different systems with different AIMC tile configurations to extrapolate performance trends given AIMC tile resource constrains. The system energy model is based on [39], [100] and shown in Table 4.5. The system energy is a combination of validated CPU active energy, CPU wait-for-memory energy, CPU idle energy, last-level cache energy, and DRAM energy. All individual AIMC tiles have 256x256 8-bit cells.

In our experiments, I aim to pick tile counts that show and extrapolate the benefits of partial network coverage without targeting any specific CNN, which may necessitate hundreds of AIMC tiles [38]. To that end, I consider three tightly-coupled systems with 2, 4, and 8 AIMC tiles per core, leading to systems that have 16, 32, and 64 AIMC tiles, respectively. I also consider three loosely-coupled systems that have 32, 64, and 96 AIMC tiles, owing to the proximity allowing for more accelerator capability.

4.6.2 Target Applications and Load Balancing

We target a variety of modern edge CNNs in this work for evaluation of our load balancing algorithms and performance benefits of partial analog coverage on the aforementioned tightly- and loosely- coupled systems. The CNNs tested includes MobileNetV2, an edge variant of SSD-ResNet34 as well as ResNet50v1.5 from the MLperf edge inference benchmark suite, and finally edge variants of VGG16 and VGG19 [65], [144], [145]. High-level layer attributes are
summarized in Tables 4.6 and 4.7, including the total number of 256x256 AIMC tiles necessary to completely cover all convolutional layers in the CNN and the total MACs per data transfer metric introduced in the hybrid digital-analog layer study. The total number of layers includes pooling and residual layers.

All CNNs are run in ALPINE for 3 to 10 inferences after cache warming (10 inferences), depending on simulation time. In general, after cache warming the inference-to-inference performance statistics have a standard deviation of less than 5%. As previously mentioned, all CNNs are implemented and optimized using tinytenslib, Eigen C++ Tensors (which are the backbone of the tensorflow library), pthreads, and ping-pong buffering to avoid blocking \cite{24,66}. Loosely-coupled variations of the CNNs are also implemented with the Taskflow library to implement work stealing. I additionally run a SIMD-enabled baseline application for each CNN that uses the aforementioned optimizations and no AIMC tiles.

In the following section, I present our results in three parts to analyze in-depth the behaviour, performance, and energy of MobileNets, Classical CNNs (VGGs), and ResNets.

### 4.7 Tightly-Coupled versus Loosely-Coupled Results

#### 4.7.1 Overall Results

A figure of all performance and energy results can be seen in Figure 4.12. A summary of the maximum speedups and energy gains for each form of coupling can be seen in Table 4.8. Note that all results are compared to a baseline system that uses the ARM NEON SIMD co-processor as well as taskflow-enabled work stealing during run time (shown using the gray dotted line in Figure 4.12).
Figure 4.12: Figure showing time (s) per inference (row 0), speedup factors (row 1), and energy improvement factors (row 2) for all tested NNs using tightly- (TC) and loosely-coupled (LC) systems with In-Order CPU models. The gray line shows the SIMD-enabled, non-AIMC tile-enabled baseline that all results are compared to. The X-axis organized by the number of tiles in the system, e.g., TC2 is two tightly-coupled AIMC tiles per CPU core, which given 8 CPU cores in the system, is 16 AIMC tiles total, and LC64 is 64 loosely-coupled AIMC tiles for the whole system. REF refers to the reference load balancing algorithm while PLL refers to the prioritize largest layer load balancing algorithm.

Table 4.8: Maximum Speedup and Energy Factors

<table>
<thead>
<tr>
<th>NN</th>
<th>TC Speedup/Energy</th>
<th>LC Speedup/Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV2</td>
<td>5.3x/5.6x</td>
<td>2.1x/2.2x</td>
</tr>
<tr>
<td>ResNet50v1.5</td>
<td>2.3x/2.9x</td>
<td>1.8x/2.0x</td>
</tr>
<tr>
<td>SSD-ResNet34</td>
<td>2.4x/3.1x</td>
<td>3.1x/3.6x</td>
</tr>
<tr>
<td>VGG16</td>
<td>2.2x/2.7x</td>
<td>5.9x/5.3x</td>
</tr>
<tr>
<td>VGG19</td>
<td>2.3x/2.7x</td>
<td>5.6x/4.1x</td>
</tr>
</tbody>
</table>

While at first glance there is no clear indicator as to how and which coupling leads to optimal performance results given resource constraints, the behaviour in performance trends can be explained by grouping CNNs into three classes: MobileNets, Classical CNNs, and ResNets. In terms of the heuristic presented in Section III (Equation 4.5), these are low MACs per transfer, high MACs per transfer, and middling-MACs per transfer classes, respectively. This heuristic turns out to accurately predict when tightly-coupled systems out-perform loosely-coupled systems, as well as when an inflection point (i.e., with what number of AIMC tiles loosely-coupled systems out-perform tightly-coupled systems) may be present, subject to the number of AIMC tiles available to the system. An explanation of how the underlying structure of these CNNs impacts their behaviour is presented in the following subsections.
4.7.2 MobileNetV2 Results and Analysis

In this work, I characterize MobileNets as a class of CNNs characterized by three primary factors. First, in comparison to other modern CNNs, they have relatively few trainable weights, usually less than 10M \cite{65}, \cite{144}, \cite{151}. Second, they leverage numerous bottleneck blocks, which are blocks of multiple point-wise and depth-wise convolutional layers. Third, the MACs per transfer rate for the whole network, when used as a "potential" for speedup, is very low (38 in the case of MobileNetV2).

This has several ramifications for usage with AIMC tiles. First, due to the lower number of trainable weights, this caps the maximum benefits that can be achieved by utilizing AIMC tiles (especially when converging to 100% AIMC tile coverage of convolutional layers). Second, because the majority of convolutional layers that can be accelerated in MobileNetV2 are point-wise (kernel size of 1), there is little data reuse, thus suggesting a high degree of data movement that can propagate an IO-based bottleneck.

It is therefore unsurprising that tightly-coupled systems greatly out-perform loosely-coupled systems in terms of energy and performance, owing to their much lower data transaction cost. I observe 5.0x-5.3x speedup and 4.8x-5.6x energy improvement in tightly-coupled systems. In comparison, I observe only 1.4x-2.4x speedup and 1.8x-2.5x energy improvement in loosely-coupled systems.

The majority of layers in MobileNetV2 can be covered by four or less 256x256 AIMC tiles, and so a high degree of speedup can be obtained early on. It is worth noting that even though the LC96 system has 96 tiles, only 94 tiles are necessary to entirely cover conventional and point-wise convolutional parameters in the network.

It is interesting to note that even a small number of AIMC tiles (as in the TC2 system) already results in a high speedup and energy improvement (5.0x and 4.8x respectively) despite the low network coverage. A layer-by-layer analysis of MobileNetV2 shows why the rate of change of speedups and energy improvement favor loosely-coupled systems.

Both REF and PLL load balancing approaches prioritize assigning AIMC tiles to three specific layers at the end of MobileNetV2: conv3, bottleneck7_1_conv2, and conv2. Applying our heuristic (equation 4.5) layer-by-layer to MobileNetV2 reveals that these layers have a maximum MACs/transfer rate of 561, 274, and 256 respectively, while all other layers have less than 137 MACs/transfer. The loosely-coupled system is able to freely assign AIMC tiles to these layers to maximize AIMC tile potential given the data throughput bottleneck. The tightly-coupled system, on the other hand, is forced to distribute more tiles to smaller layers where there is less potential for AIMC tile-based speedup and energy gains. As a result, many small layers become localized in MobileNetV2, resulting in more layers receiving AIMC tiles and a reduction in the overhead of cross-core communications.

In summary, when the MACs per transfer rate of a CNN is low, as is the case in MobileNetV2,
the network’s primary bottleneck is data transactions. AIMC tiles are then best placed in tightly-coupled configurations, where even despite the more constrained area available, can easily out-perform loosely-coupled systems even when less AIMC tiles are available. Loosely-coupled systems can leverage the more available area to pipeline layers within the AIMC cluster to try and resolve data movement-based bottlenecks.

4.7.3 VGG Results and Analysis

If MobileNets represent networks that are best accelerated by tightly-coupled AIMC tiles, then "classical" CNNs represent the opposite case. I characterize these CNNs as having larger layers (i.e., with more trainable weights) towards the end of the network pipeline and layers that commonly have larger kernel sizes. It is worth noting that based on our heuristic MACs/transfer rate, VGG neural nets have the largest MACs/transfer values.

For VGG16, I observe 2.0x-2.2x speedup and 2.3x-2.7x energy gains in tightly-coupled systems and 1.4x-5.9x speedup and 1.5x-5.1x energy gains in loosely-coupled systems. For VGG19, I observe 2.2x-2.3x speedup and 2.3x-2.7x energy gains in tightly-coupled systems and 1.2x-5.6x speedup and 1.2x-4.1x energy gains in loosely-coupled systems. REF load balancing out-performed PLL in VGG16 and vice-versa in VGG19.

To explain VGG’s performance trends, it is worth noting that VGG16 and VGG19 have six and eight layers, respectively, at the end of the network pipeline with an im2col transformed height of 4608 and width of 512. conv9 and conv10 in VGG 16 and then conv10, conv11, and conv12 in VGG19 specifically, are all primarily targeted by the load balancing algorithms to have 37M MACs, each, per inference.

For AIMC tile-enabled systems, this represents a simultaneous challenge and opportunity to properly allocate resources, given that each of these layers would require 36 256x256 AIMC tiles. The loosely-coupled systems are able to freely allocate 36 tiles to these layers thanks to the flexible access to the AIMC cluster interface. However, the best-case scenario for tightly-coupled systems is that they assign the AIMC tiles they have to these layers and then perform the rest of the computation with digital (CPU) logic. Recall that in Section III, for large convolutional layers I showed that it is difficult to attain very high speedups without a high degree of layer coverage. Tightly-coupled systems then are only able to gain speedups by distributing more tiles to other high-MAC layers to balance CPU workloads.

This same issue with layer coverage overhead also ends up explaining the trade-offs between distributed AIMC tile-based load balancing (REF) and centralized AIMC tile-based load balancing (PLL). In our VGG16 experiments with 64 loosely-coupled AIMC tiles, 22 AIMC tiles are assigned to conv9 and conv10 using REF, representing representing 61% layer coverage – in other words, speedup is likely. However, in PLL, conv9 gets 36 AIMC tiles while conv10 only obtains six (note other AIMC tiles are assigned to other layers just to balance CPU workloads). This means that in PLL, while conv9 would enjoy an exponential speedup benefit, conv10
would likely not result in any speedup due to interface overhead. This is why experimentally, REF performs better than PLL in VGG16. VGG19 has more balanced coverage beyond 50% thresholds for AIMC tile allocations, hence why PLL generally performs better. For further improvements to AIMC tile load balancing, it is suggested that thresholding is used to prevent low AIMC tile allocations to large layers.

In summary, due to numerous large layers with high data reuse factors, loosely-coupled systems tend to excel over tightly-coupled system when running "classical" CNNs like VGG due to their ability to freely allocate as many AIMC tiles as necessary or until resources are exhausted. However, as shown with the REF and PLL load-balancing algorithms, a low degree of coverage as an artifact of automated load balancing algorithms can also result in worse performance. It is therefore advisable to modify such algorithms to utilize thresholds to ensure that accelerator overheads are overcome such that speedup and energy benefits remain large.

4.7.4 ResNet Results and Analysis

In comparison to MobileNets and "classical" CNNs, ResNets represent a sort of compromise. Residual blocks tend to be made of both large convolutional layers with larger kernels as well as point-wise convolutions. The MACs per transfer rate of ResNet50v1.5 and SSD-ResNet34 are middling in comparison to the other NNs tested in this work, having values of 203 and 580, respectively. Based on this heuristic alone, I would predict that ResNet50v1.5 performs better in tightly-coupled architectures and SSD-ResNet34 performs better with loosely-coupled architectures, however, our results actually suggest that there is an inflection point in the performance of differently coupled systems. Additionally, it is worth noting that our maximum speedups are capped by the fact that both CNNs require over 350 AIMC tiles to cover all convolutional layers completely.

For ResNet50v1.5, I observe 2.0x-2.3x speedup and 2.2x-2.9x energy gains in tightly-coupled systems and 1.3x-1.8x speedup and 1.5x-2.0x energy gains in loosely-coupled systems. For SSD-ResNet34, I observe 2.2x-2.4x speedup and 2.4x-3.1x energy gains in tightly-coupled systems and 2.0x-3.1x speedup and 2.0x-3.6x energy gains in loosely-coupled systems. Extrapolating the trends in performance for both NNs, an inflection point in performance gains (i.e., loosely-coupled performs better than tightly-coupled) appears around 200 AIMC tiles and 75 AIMC tiles for ResNet50v1.5 and SSD-ResNet34, respectively.

In general, both REF and PLL will target the middle convolutions in residual blocks that contain large kernel sizes, and therefore large data reuse factors. Owing to the slightly smaller sizes of layers (in comparison to VGG16 conv9 and conv10, for example), PLL is able to more adequately use tiles to obtain an average 20% speedup and energy gain over REF load balancing.

In summary, AIMC tiles are best allocated in ResNets to the layers that have larger kernel values in order to leverage data reuse. Their middling MACs per transfer value accurately
predicted an inflection point in tightly- and loosely-coupled system performance that can be leveraged before inference run time.

### 4.8 Summary

In this chapter, I performed a large study of how to extract performance and energy benefits from edge AI AIMC tile-enabled solutions, given the trade-offs in network size, available AIMC tile resources, and configurations of AIMC tiles and clusters.

I performed an initial single-layer study of hybrid digital-analog layers, which showed that there exists a threshold from where I can attain benefits from AIMC tiles. I used this study to develop a heuristic metric, MACs per transfer rate, to guide and explain the performance behavior of different AIMC tile allocations.

By then implementing a loosely-coupled accelerator model that utilizes the packet interfaces of the ALPINE generic PIO device, I am then able to perform an architectural exploration that compares interfaces for differently-coupled accelerators of varying resource amounts.

I then explore the performance and energy benefits of two different load balancing techniques – distributed and priority for largest layers – for both CPUs and AIMC tiles in multi-core systems. Using five real edge CNNs, classed as MobileNets, ResNets, and classical CNNs, our analysis shows that tightly-coupled AIMC tiles, despite likely being more limited in resources, are better for NNs that exhibit low data reuse and, therefore, a low MACs per transfer rate due to their very low data transfer overhead, while loosely-coupled AIMC clusters are better for classical CNNs due to their ability to more freely allocate more AIMC tiles to larger layers in order to meet thresholds that enable great performance gains. I ultimately show up to 5.9x speedup and 5.6x energy gains even in cases where AIMC tiles cover a small portion of the NN.
5 Conclusions

As AI and machine learning-enabled applications continue to pervade society and shape the world we live in, their utility is subject to the need to reduce energy and make societies more sustainable. The endlessly-increasing compute and memory intensities of AI applications combined with the energy and form factor constraints of edge devices means new architectures that challenge the Power and Memory Walls must be explored, lest we face a stagnant future that is hindered by lack of innovation. Architectural explorations that target specific dimensions of heterogeneity are key to building novel architectures that address not only edge AI applications’ needs now but those in the future as well.

Therefore, in this thesis, I have presented the ALPINE framework, which expands the dimensions of heterogeneous architecture explorations to integrate multiple acceleration techniques and interfaces.

5.1 Summary of Contributions

5.1.1 gXR5 and ALPINE

In Chapter 2 of this thesis, I presented the engineering work behind gXR5 and ALPINE, as well as the validation research effort that went into gXR5.

With gXR5, we expand the gem5 simulator to support validated RISC-V architectures, achieving the world’s first known full system-level, Linux-capable simulation of RISC-V systems. This effort brought the RISC-V privileged specification to gem5, including new RISC-V ISA instructions, interrupter models, a MMU, and the full software stack for full system-level simulations. gXR5 was then validated against a real RISC-V system using stress-ng microbenchmarks and select benchmarks from the SPEC CPU2017 suite, reducing the mean validation error to 24%.

gXR5 set up the basis for ALPINE, which expanded gem5-X by integrating a generic peripheral input-output model that can be easily configured to model a variety of accelerators with different interfaces. ALPINE links the low-level hardware model with high-level software by
Chapter 5

Conclusions

introducing multiple libraries that can leverage both kinds of interfaces through a simple compiler flag and easily spin up fast and optimized neural networks.

gXR5 and ALPINE both set up the basis for edge AI architecture explorations.

5.1.2 Tightly-Coupled Edge AI Solutions

To showcase ALPINE’s capabilities, I first introduce the concept of accelerators tightly-coupled to their CPUs in proximity and interface, and implement an analog in-memory compute (AIMC) model as a first test case.

By leveraging analog logic to perform matrix-vector multiplication operations (MVMs) in constant time, I am able to overcome the primary bottleneck in machine learning algorithms, as shown for MLPs, RNNs, and CNNs. I perform three exploration studies with ALPINE on the aforementioned networks, looking in-depth at how to map neural networks to the AIMC tiles in single-core and multi-core systems, as well as analyzing regions-of-interest in AI inference to extrapolate the speedup and energy benefits of AIMC tile-enabled systems. I showed how AIMC tiles target the primary bottleneck in MLPs by reducing MVMs from 96% of the average run time of the application to less than 1% of the total run time, while still obtaining over 8x speedup and energy gains over all. The MLP exploration study also first shows cross-core communication as an emerging bottleneck in MPSoCs. The RNN exploration study then used LSTMs to show the scalability of AIMC tiles when applied to networks of varying sizes, proving the scalability of compute and memory complexities. Then finally a study of three multi-core VGG8 variant CNNs with AIMC tiles yielded over 20x speedup and energy gains while revealing the role of layer-by-layer bottlenecks in MPSoCs in blocking further performance gains.

To show the flexibility of ALPINE, I then contributed to an effort to refactor the generic PIO device in order to model tightly-coupled systolic arrays (TiC-SATs), a competing edge AI solution. Using Transformers as the primary case study, we showed up to 89.5x speedup in the BERT-large transformer, as well as massive speedups in a huge variety of BERT and vision transformer models by leveraging systolic arrays for tiling matrix-matrix multiplication operations (MMMs) and reducing data transaction overheads.

With two competing edge solutions now available in ALPINE, I achieve one of the goals of this thesis by performing the world’s first direct experimental comparison of AIMC and systolic array-based technologies for a common edge AI application – the same aforementioned VGG8 variants. I introduce area and chip form factor as a new constraint in order to fairly compare the nuances of the implementations and algorithms of these competing solutions to each other, as well as to SIMD-enabled systems and systems that leverage larger 16x or 32x more cache. This study reveals the role of the simultaneous computation and memory bottlenecks present in neural networks, showing that more caches are ineffective in improving performance due to the size of the working set. I show how a lack of data reuse in convolutional layers with large kernels leads to performance hindrance of CNNs with TiC-SATs, while rearranging AIMC
Conclusions Chapter 5

Tiles for optimal layer coverage yields up to over 9.4x speedup and energy improvement over systems that simply use more caches.

5.1.3 Loosely-Coupled Edge AI Solutions

To expand upon the dimensions of heterogeneity with which ALPINE can be used to conduct edge AI architectural explorations, I built upon the tightly-coupled generic PIO model to leverage a loosely-coupled, bus-bounded interface. With the infrastructure in place, I am able to perform a first-of-its-kind architectural exploration that examines the trade-offs in edge AI accelerator proximity and resources.

Using AIMC tiles as my primary study target, I first performed experiments on hybrid analog-digital fully connected and convolutional layers to determine what allocation of constrained resources leads to the best results. I showed that while 102x speedup is attainable for a fully-covered 4096x4096 dense layer, the resource cost is extremely high, requiring 256 256x256 AIMC tiles. Furthermore, because of data transaction overheads, speedups are only attainable once over 50% of the layer’s weights reside in the AIMC tiles, suggesting that fully connected layers are not the most optimal targets for AIMC-based acceleration.

Then, I selected a variety of convolutional layers based on their im2col transformations from MobileNetV2, VGG16, and ResNet50v1.5 to carry out the hybrid computation study. By leveraging data and input reuse, I was able to gain up to 17.4x speedup with a fully-covered convolutional layer with only 18 256x256 AIMC tiles. My analysis of this study led to the proposal of a new metric, MACs per transfer, to predict the benefits of applying AIMC tiles to different convolutional layers by quantifying the number of MACs performed in constant time versus the number of data transactions that need to occur moving input and outputs to and from the AIMC tile, respectively.

This study set up the basis for our loosely-coupled AIMC cluster comparison to tightly-coupled AIMC tile-enabled MPSoCs. I proposed two load balancing strategies, reference (REF) and prioritize largest layer (PLL), that simultaneously balance CPU workloads and AIMC tile resources, and then apply them separately to both loosely-coupled and tightly-coupled systems. Using five modern edge-class CNNs, I showed how the MACs per transfer rate of the whole CNN can accurately predict when loosely-coupled systems can out-perform tightly-coupled systems by classifying the neural network as having high, low, or middling MACs per transfer rate, which correlate to classical CNNs, MobileNets, and ResNets, respectively. A NN with a low MACs per transfer rate was shown to always be data transaction-bound, and is therefore always optimal on tightly-coupled architectures, while a NN with high MACs per transfer rate is bound by computation rather than data movement, and is therefore better on loosely-coupled architectures even when the data transaction overhead is high. I ultimately show up to 5.9x speedup and 5.6x energy gains even in cases where AIMC tiles only cover 42% of the NN’s weights.
5.2 Future Work

Over the course of my work in developing this thesis, numerous other dimensions of heterogeneity and state-of-the-art solutions have presented themselves as prime candidates for further exploration. In this section, I present other works that can be expanded upon to further enhance the ALPINE framework.

5.2.1 Overcoming Communication Bottlenecks with Wireless Scratchpad Memories

The initial case studies of tightly-coupled AIMC tiles within the ALPINE framework (Section 3.3) revealed cross-core communication as an emerging bottleneck. More specifically, less than 1% of a MLP’s run time, when accelerated with AIMC tiles, is dedicated to MVMs and over 70% of the run time is dedicated to communicating with other CPU cores also performing accelerated MVMs. This was shown to hinder application performance when the application became more parallel (going from single-, to dual-, to quad-CPU core). One idea for overcoming this emerging bottleneck is to employ technologies that optimize the communication between CPU cores as a means of achieving further speedup and energy gains. A solution that has been gaining traction in the literature is short-distance wireless nano-transceivers that perform wireless communication in between chips in large MPSoCs [152].

These small wireless transceivers have been demonstrated to have up to 120Gbps bandwidth [153]. Therefore, I worked with Rafa Medina on the EU Horizons 2020 Project’s WiPLASH initiative to implement wireless scratchpad memory models into ALPINE. The goal was to model systems that could model both AIMC tiles and wireless communications in MPSoCs jointly. By providing the ALPINE framework, I contributed to a work by Rafa Medina that models wireless scratchpad memories in ALPINE. These scratchpad memories exist close to the CPU and are meant to facilitate CPU-to-CPU communications. We showed up to 2.64x speedup in deep neural networks that employed wireless scratchpad memories alone. This work was published in ASP-DAC 2023 [154].

We then jointly tested CNN performance on an architecture that employs both wireless scratchpads and AIMC tiles. We used the same VGG8 variant CNNs as tested in Section 3.3.7, from [109] on the same high-power system architecture used in Section 3. The largest of the CNNs performed 20% more efficiently with both AIMC tiles and wireless scratchpads, leading to a maximum 25.6x speedup (up from 20.5x speedup from AIMC tiles alone). This effort shows how the use of multiple heterogeneous edge AI solutions together can significantly increase performance.
5.2.2 Coherent Accelerators and Overcoming Address Translation Bottlenecks with Midgard

To make accelerators more independent from CPUs, they can be made coherent by integrating a coherent bus (e.g. CXL) and MMU which then allows them to access caches and memory directly. An issue that quickly arises from coherent accelerators, however, is address translation overhead, which in certain environments can be responsible for up to 30% of performance overhead [155].

Introduced in ISCA 2021, Midgard is an intermediate address space proposed to reduce address translation overheads in cache hierarchies [156]. Midgard introduces CPU core-side access control of virtual memory areas (VMAs) and memory-side page-granularity translation to lower the overhead of access to caches in large-data, cache-resident workloads. The result is that virtual-to-Midgard address translation overhead can fall to 0% for cache-resident workloads, while virtual-to-memory address translation overhead is only 5% more for non-cache-resident workloads.

Therefore, if we imagine a heterogeneous addressing system that employs both a traditional MMU on the CPU and a Midgard-enabled MMU on an accelerator, it may be possible to lower address translation overheads for the accelerator at no overhead cost to the CPU. This would require expanding ALPINE by integrating Midgardian translation into the MMU model, and then implementing a CXL interface on top of the ALPINE PCI model. The generic PIO device introduced in ALPINE can be augmented with the new MMU model, once a prototype is validated and working. This effort is currently being undertaken by Qunyou Liu in ESL, EPFL, and is apart of a submission to HiPEAC 2024.

5.2.3 Validated Full System-Level GPU Modeling

While GPUs are often considered state-of-the-art for server-class AI training and inference, they remain relatively uncommon to see in edge domains due to extremely high energy requirements. With the introduction of the Nvidia Jetson platform as well as integrated GPUs in advanced edge systems like the Apple iPhone however, we are already starting to see GPUs at the edge [157], [158].

Therefore, to continue being able to perform heterogeneous explorations targeting novel edge AI architectures, it may be pertinent to implement a validated GPU model into gem5. Unfortunately, at time of writing, there is no publicly validated GPU model for gem5 that can run full system mode, thus restricting this dimension of heterogeneity for future architectural explorations. gem5-21 introduced an AMD Graphics Core Next (GCN) architecture model, which includes support for GPU thread contexts, GPU compute unit pipelines, and the ROCm software stack, but it is not validated against real hardware and only runs in software emulation (SE) mode [26].

Implementing a validated GPU model in ALPINE and gem5 would be a monumental effort,
including the creation, implementation, and validation of numerous new models, including
GPU compute cores, memories, caches, interconnects, services, and more. However, the end
result would be great: first, an implemented and validated GPU model representative of the
state-of-the-art would provide another baseline for which modern edge AI solutions could be
compared. Second, heterogeneous GPU architectures that can employ modifications to both
the GPU and whole system architecture, could then be explored.
Bibliography


[34] K. Asanović and D. A. Patterson, “Instruction sets should be free: the case for risc-v”, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014.


[100] Y. M. Qureshi et al., “Gem5-x: a gem5-based system level simulation framework to optimize many-core platforms”, in Spring Simulation Conf. (SpringSim), 2019.


Joshua Klein
Diestsesteenweg 525, 3010 Kessel-Lo, Vlaams-Brabant BE
+32 470 32 03 31
joshuaahklein@gmail.com

Researcher in the field of Electrical/Computer Engineering with extensive project management and hands-on research experience in embedded systems, RISC systems, software engineering, applying machine deep learning applications, and heterogeneous computer architectures using cycle-accurate simulation and real hardware.

Skills
• **Computer:** gem5/cycle-accurate simulation and modeling, scripting (shell), Qemu, GitHub, GitLabs, Microsoft Office, Google Drive, LaTeX, notion.so.
• **Operating Systems:** Mac OS X, Ubuntu, CentOS.
• **Programming Languages:** Python (matplotlib, scikit-learn, tensorflow), C, C++ (Eigen, Taskflow, pthreads). Also have experience with C# .NET, ASP.NET, Java, Verilog HDL, CUDA.
• **Microprocessors:** HiFive SoCs, Raspberry Pi, Arduino, Beaglebone Black.
• **Languages:** English (native), French (B1).

Education
**École Polytechnique Fédérale de Lausanne, Switzerland**
Doctoral Student of Electrical Engineering
13 ECTS Credits

December 2019 – October 2023

**École Polytechnique Fédérale de Lausanne, Switzerland**
Internship
July 2019 – December 2019

**Boston University College of Engineering, Boston, MA**
Master of Engineering in Computer Engineering
GPA: 3.9/4.0

May 2019

**Boston University College of Engineering, Boston, MA**
Bachelor of Science in Computer Engineering, Magna Cum Laude
GPA: 3.74/4.0 (Dean’s List – 5 Semesters)
Member of IEEE-HKN

May 2017

Research Projects and Publications

"Which coupled is best coupled? An Exploration of AIMC Tile Interfaces and Load Balancing for CNNs", **EPFL Embedded Systems Laboratory and IBM Research Zürich**

July 2022 – October 2023

- Expanded upon gem5-X + ALPINE simulation framework to integrate “loosely-coupled” interfaces for IBM Zürich’s analog in-memory computing model.
- Coordinated with HEIG-VD REDS Institute for development of OS utilities for loosely-coupled accelerator model.
- Developed a custom Eigen C++-backed library that expands upon ALPINE’s AIMCl library to quickly develop apparatus-ready tensor networks for performance and energy modeling experiments.
- Performed hybrid digital-analog performance experiments to quantify the scalability of partially-accelerated workloads in resource-constrained systems.
- Analyzed novel and state-of-the-art load-balancing algorithms for MPSoC-backed CNNs in the case of differently-coupled systems.
- Submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS).

“gXR5: Integration and validation of RISC-V full system-level simulation into the gem5 computer architecture simulator”, **EPFL Embedded Systems Laboratory**

July 2019 – November 2022

- Created the first known full Linux-capable system-level simulation framework for RISC-V systems in the gem5 simulator, including the ability to load bootloader, OS, and cross-compiled binaries into a simulated RISC-V system.
- Developed and tested gem5 models for numerous hardware components, including interrupters, timers, and RISC-V compliant MMU.
- Coordinated and managed collaborators at HEIG-VD as well as M.Sc. students at EPFL for validating gXR5 performance statistics and models against real RISC-V hardware.
- Maintained software toolchain, repositories, and documentation.
• Headed tutorial, “Using gem5 and full-system RISC-V simulation to enable the optimization of heterogeneous architectures”, HiPEAC 2021.
• Submitted to DAC 2024: Karan Pathak, Joshua Klein et al., "Towards Accurate RISC-V Full System Simulation via Component-level Calibration”.

“ALPINE: Analog In-Memory Acceleration with Tight Processor Integration for Deep Learning”,
EPFL Embedded Systems Laboratory and IBM Research Zürich
January 2020 - Present

• Developed simulation framework for exploring and evaluating computer architectures with tightly-integrated machine learning accelerators in the gem5-X full-system level simulator.
• Implemented, tested, and calibrated IBM Zürich’s analog in-memory computing model into the ALPINE framework.
• Created a C++ software library for quickly implementing and validating deep learning applications in the ALPINE framework.
• Used POSIX threads and Eigen 3.8 to design, develop, and highly-optimized single- and multi-core deep learning applications in both pure digital and accelerator-enabled variants for performance comparisons, including MLPs, RNNs, and CNNs.
• Responsible for project documentation, results visualization, and timeline management.

Other Publications:
• Morillas, Rafael., et al. “System-Level Exploration of In-Package Wireless Communication for Multi-Chiplet Platforms”, in 26th Asia and South Pacific Design Automation Conference (ASP-DAC). 2023

Masters Project
“Determining Optimal Memory Allocation Thresholds in MOCA for Heterogeneous Memory Systems”,
Boston University Peaclab September 2018 – December 2018

• Used and modified gem5 to model thresholds for a variety of simulated heterogeneous memory architectures under the MOCA framework.
• Developed methods for determining optimal placement of memory objects in heterogeneous main-memory systems.
• Verified energy/performance results via gem5 models, Micron power calculators, and McPat.
• Implemented shell and python scripts for large-scale data organization and parsing.

Undergraduate Senior Project
“Telescope Mount for Tracking LEO Satellites”, MIT Lincoln Labs September 2016 – May 2017

• Worked in tandem with a five person team to create a prototype base station telescope mount for optical communications with low-earth orbit (LEO) satellites.
• Created GUI and interfaced software control system with C#.NET.
• Awarded “Best ECE Senior Design Project 2016/2017” by Boston University.

Professional Experience
Memory Sub-systems Researcher, imec November 2023 – Present

• Working to evaluate the area, performance, and energy of full system-level memory sub-systems using novel memory technologies as well as compute-near-memory and compute-in-memory.
• Responsible for building and maintaining full system-level framework and infrastructure for architectural explorations.
• Work with cycle-accurate modeling tools such as gem5 and Ramulator.
• Coordinate memory characterizations and models with imec technology teams and partners.

Technical Consultant, Bryson Editorial  
December 2018 – May 2019
• Automated editorial data processing to streamline workflow.
• Provided technical expertise for reviews of technical articles and internal documentation.

Boston University Research Assistant  
October 2017 – May 2019
• Managed and worked with a small team to build upon a prototype autonomous boat swarm that tracks drug submarines via underwater vibrations in low SNR environments.
• Worked on the scalability of the autonomous boat system, including software and hardware integration for Beaglebone Blacks, Pixhawks, Raspberry Pis, Navio2s, and XBees.
• Implemented ROS controls and interface for the autonomous boat system.
• Prototyped whole system and implemented unit tests of hardware and software modules.
• **Spoke at NEEC Day 2018, “Swarms of Autonomous Boats”**.

Boston University Teaching Assistant  
September 2015 – May 2019
• Acted as teaching assistant for undergraduate and graduate courses.
• Responsibilities included creating grading and solutions frameworks, developing auto-grading scripts, providing tutoring hours to students, and heading lab sections.

Related University Projects
• **SatLight Satellite Tracking Utility**: Research, designed, and implemented a stationary satellite tracking utility using a Gumstix board and custom image processing algorithm.
• **Autonomous Rover with Localization and Collision Avoidance**: Worked in tandem with and managed a five-person team to design and implement an autonomous vehicle. Headed system design and integration, localization, PID, and database API using SQLite and Python.

University Courses