Thèse n° 9261

In-Memory Hardware and Architectural Extensions for Workloads Acceleration

Présentée le 7 septembre 2022

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

William Andrew SIMON

Acceptée sur proposition du jury

Prof. G. De Micheli, président du jury
Prof. D. Atienza Alonso, directeur de thèse
Prof. I. O'Connor, rapporteur
Dr A. Rahimi, rapporteur
Prof. A. Burg, rapporteur

École
polytechnique
fédérale
de Lausanne

2022

Thirst was made for water; inquiry for truth.
— C.S. Lewis

Dedicated to everyone who got me here,

Acknowledgements

THE last 4+ years have been some of the most exciting, formative, and challenging years of my life thus far. This is due in no small part to the people who I have had the honor to call my friends and colleagues, who have rejoiced with me in the highs and supported me in the lows. I am fully convinced that I could not have gotten to this point without their technical and/or emotional support. There is no way I can give everyone the proper recognition they deserve without this section being longer than the thesis proper, so I apologize in advance for my forgetfulness and generalizations; know that you are all appreciated and loved.

First of all, I would like to thank my jury members for graciously volunteering their time to read my thesis and provide valuable feedback. Specifically, I'd like to thank Prof. **Giovanni de Micheli**, who also supervised me during two Master's projects as well as attending my candidacy exam. I'd also like to thank Prof. **Andreas Burg**, who also attended my candidacy exam, and who has been a helpful colleague, both directly and indirectly through his excellent students in the Telecommunications Circuits Laboratory who I have had the good fortune of interacting with. Finally, I would like to thank Dr. **Abbas Rahimi** and Prof. **Ian O'Connor**, who don't know me from Adam but were more than happy to be external members of my jury.

Then, it is essential for me to thank Prof. **David Atienza**, without whom none of this would have been possible. Throughout my time here, I have seen him go above and beyond what is expected of any professor in terms of supporting my work as well as the rest of the members of ESL. How he manages to maintain a real-time knowledge of the status of 30+ PhD students to the degree that he is able to make such meaningful contributions at our bi-weekly meetings is beyond me. Aside from his academic support, he also worked hard to build lab cohesion through many social events, including lab dinners, ski trips, paintball games, and races. A lab is only as good as its professor, and David has made ESL a fun and rewarding place to work. I can't thank him enough.

Of course, while David was more than happy to help out at any time, I didn't feel at liberty to pester him every time I ran into a problem. That role fell to my amazing post-doctoral advisors. First, Prof. **Marina Zapater** has been with me from the start, helping me to sharpen my initial visions into concrete actionable ideas. More than once she helped me through points when I felt like packing up my desk and being done. Next, it isn't an understatement to say that the expertise of Dr. **Alexandre Levisse** made this PhD possible. Alex joined the lab at the exact moment he was needed most, and without him this work would have looked very different and in all likelihood not nearly as successful. Finally, Dr. **Giovanni Ansaloni**, although late to the party (mine at least, he was in ESL years before me), has contributed immensely to the last year's work, effortlessly coming up with clever paper titles, extracting new insights from data, and contributing valuable proofreading feedback. These three people have been amazing mentors, but of course they aren't the only post-docs in the lab. I'd like to thank all of our mentors for their various contributions to our work; in particular, Drs. **Miguel Peón**, **Adriana Valdes**, **Tomás Teijeiro**, and **Luis Costero** have been good friends on this journey.

I'd also like to thank all of the members of ESL who made the time here so enjoyable. It's been so great being a part of such a diverse group of people who love to work hard and play hard. Thanks to **Elisabetta de Giovanni, Grégoire Surreal, Artem Andreev, and Renato Zanetti** for all the adventures around Switzerland. Thanks to **Marco Rios and Yasir Qureshi** for being hardworking colleagues, co-authors, and supportive friends. Thanks to **Homeira Salimi** for keeping all of our office related logistics in order and planning the various ESL activities. Thanks to **Rodolphe Buret, Mikäel Doche, and Christoph Müller** for keeping our servers running despite our best efforts to break them.

And finally, thanks to all the rest of the lab for the constant interesting lunch and coffee conversations, team building runs, and all-round general fantastic work environment.

Of course, work only represents half of my life here. Outside of the office I am indebted to so many people who have made my life in Switzerland a joy. I am forever indebted to the community of Westlake Church, among whom I count my oldest and closest friends in Switzerland. The countless evening and weekend events served to break up the sometimes monotony of work and refreshed my mind and soul. I would like to especially thank the **Slack** family, who adopted me during my time here and whose couch I slept on numerous times on lazy Sunday afternoons. I also want to thank Prof. **Thomas Rizzo** and his wife **Karen**, who have put much effort into connecting Christian academics on EPFL's campus through lunches and dinners. And of course, the young adults group contains too many names to mention. I started to list you, but really its impossible. I love you all so much.

Finally, a few special people I'd like to mention. None of this would have been possible without my incredibly supportive family, **Brad, Kelly, Jonathan, Brianna, and Kermit**, who I could always call upon for advice or just to talk about life. I owe them so much for pushing, encouraging, and supporting me in overcoming challenges both inside and outside of academia. Finally, I'd like to make a special thanks to **Una Pale** for being such an incredible person over the last year and some odd months. Thank-you so much for putting up with the sometimes tears and existential crises, as well as always being down for a weekend adventure or a cross country trip. Volim te bobice moja!

Lausanne, May 28, 2022 William Simon

Abstract

Utilization of edge devices has exploded in the last decade, with such use cases as wearable devices, autonomous driving, and smart homes. As their ubiquity grows, so do expectations of their capabilities. Simultaneously, their form factor and use cases limit power availability. Thus, improving performance while limiting area and power consumption is paramount.

In this vein, in-Memory Computing (iMC) moves computation from the CPU into the memory hierarchy. This has multiple benefits. First, reduced data movement mitigates power consumption and latency. Second, the entire memory array can be utilized to perform hundreds of concurrent operations.

iMC has been proposed in various technologies, such as in SRAM, DRAM, and emerging Non-Volatile Memories (eNVRAMs). In particular, in-SRAM Computing (iSC) benefits from integration into existing products such as cache hierarchies, reducing area overhead and implementation complexity. Further, iSC operations are generally digital in nature, as opposed to analog, precluding the need for costly ADCs and improving application accuracy. Conversely, iSC accelerators face challenges in avoiding memory corruption, running at high frequencies, and kernel/application level integration.

This thesis exploits iSC while addressing the aforementioned challenges via a BitLine Accelerator for Devices on the Edge (BLADE). BLADE can be implemented in any SRAM system and utilizes local wordline groups to perform computations at a frequency 2.8x higher than state-of-the-art iSC architectures. BLADE is thoroughly simulated, fabricated, and benchmarked at the transistor, architecture, and software abstraction levels. Experimental results demonstrate performance/energy gains over an equivalent NEON accelerated processor for a variety of edge device workloads, namely, cryptography (4x performance gain/6x energy reduction), video encoding (6x/2x), and convolutional neural networks (3x/1.5x), while maintaining the highest frequency/energy ratio (up to 2.2Ghz@1V) of any conventional iSC computing architecture, and a low area overhead of less than 8%.

With BLADE implemented, the possibilities for enhancement are manifold, with one such example being approximate computing. To this end, a CArryless Partial Product InExact Multiplier (CAPPIEM) halves multiplication latency while incurring negligible area overhead. As a standalone multiplier, CAPPIEM reduces the area/power-delay-product by 73/43%, respectively. Further, CAPPIEM has the unique property of computing exact results when one input is a Fibonacci encoded value. This property is exploited via a retraining strategy which quantizes neural network weights to Fibonacci values, ensuring exact computation during inference. Benchmarking on Squeezenet 1.0, DenseNet-121, and ResNet-18 demonstrate accuracy degradations of only 0.4/1.1/1.7%, while improving training time by up to 300x.

A second BLADE enhancement is the use of SRAM/eNVRAM Hybrid Caches (HCs). HCs increase capacity and power savings via eNVRAM's small area footprint and low leakage energy. However, eNVRAMs also incur long write latency and limited endurance. In this context, this thesis presents SHyCache, an HC architecture and supporting programming model. Variables with high read/write access ratios can be explicitly allocated to the eNVRAM array. This reduces access time, power consumption, and area overhead while maintaining maximal utilization efficiency and ease of programming. Benchmarks on a range of cache hierarchy variations using three deep neural networks demonstrate a design space that can be exploited to optimize performance, power consumption, or endurance, while demonstrating maximum performance and power gains of 1.7/1.4/1.3x and 5.1/5.2/5.4x, respectively.

Keywords: *in-memory computing, in-cache computing, machine learning, artificial intelligence, neural networks, approximate computing, edge computing, edge AI, computing architectures, hybrid caches, SRAM memory, emerging non-volatile memories*

Résumé

L'UTILISATION des dispositifs périphériques a explosé au cours de la dernière décennie, avec des cas d'utilisation tels que les dispositifs portables, la conduite autonome et les maisons intelligentes. Leur omniprésence augmente, tout comme les attentes concernant leurs capacités. Simultanément, leur facteur de forme et leurs cas d'utilisation limitent la disponibilité de l'énergie. Il est donc primordial d'améliorer les performances tout en limitant la surface et la consommation d'énergie.

Dans cette optique, le calcul en mémoire déplace le calcul de l'unité centrale vers la hiérarchie de la mémoire. Cela présente de multiples avantages. Premièrement, la réduction des mouvements de données entre le CPU et la mémoire réduit la consommation d'énergie et la latence. Ensuite, la largeur totale de la mémoire peut être utilisée pour effectuer des opérations SIMD massives.

Le calcul en mémoire a été proposé dans diverses technologies, comme la SRAM, la DRAM et la eNVRAM. En particulier, le calcul en SRAM bénéficie de l'intégration dans des produits existants tels que les hiérarchies de cache, ce qui réduit la surcharge de surface et la complexité de mise en œuvre. En outre, les opérations in-SRAM sont généralement de nature numérique, par opposition à l'analogique, ce qui évite le recours à des convertisseurs analogiques-numériques coûteux et améliore la précision des applications. À l'inverse, les accélérateurs in-SRAM sont confrontés à des difficultés pour

éviter la corruption de la mémoire, pour fonctionner à des fréquences élevées et pour intégrer les niveaux noyau/application.

Cette thèse exploite le calcul in-SRAM tout en relevant les défis susmentionnés via BLADE. BLADE peut être implémenté dans n'importe quel système SRAM et utilise des groupes de lignes de mots locaux pour effectuer des calculs à une fréquence 2.8x plus élevée que les architectures de calcul in-SRAM les plus récentes. BLADE est soigneusement simulé, fabriqué et évalué aux niveaux du transistor, de l'architecture et de l'abstraction logicielle. Les résultats expérimentaux démontrent des gains de performance/énergie par rapport à un processeur accéléré NEON équivalent pour une variété de charges de travail de dispositifs périphériques, à savoir la cryptographie (gain de performance de 4 fois/6 fois la réduction d'énergie), le codage vidéo (6x/2x) et les réseaux neuronaux convolutifs (3x/1,5x), tout en maintenant le rapport fréquence/énergie le plus élevé (jusqu'à 2,2 GHz@1V) de toute architecture de calcul in-SRAM conventionnelle, et un faible surcoût de surface de moins de 8

Avec la mise en œuvre de BLADE, les possibilités d'amélioration sont multiples, l'un de ces exemples étant le calcul approximatif. À cette fin, CAPPIEM réduit de moitié la latence de la multiplication tout en introduisant un surcoût de surface négligeable. Lorsqu'il est mis en œuvre en tant que multiplicateur autonome, CAPPIEM réduit le produit surface/ puissance-retard de 73/43%, respectivement. En outre, CAPPIEM a la propriété unique de calculer des résultats exacts lorsqu'une entrée est une valeur codée Fibonacci. Cette propriété est exploitée par l'introduction d'une stratégie de réapprentissage qui quantifie les poids du réseau neuronal en valeurs codées Fibonacci, garantissant un calcul exact pendant l'inférence. L'évaluation comparative sur Squeezenet 1.0, DenseNet-121 et ResNet-18 montre une dégradation de la précision de seulement 0,4/1,1/1,7%, tout en améliorant le temps de formation jusqu'à 300 fois.

Une deuxième amélioration de BLADE est l'utilisation de caches hybrides composés de cellules binaires SRAM et eNVRAM. Les caches hybrides augmentent la capacité et les économies d'énergie grâce à la faible empreinte de la eNVRAM et à sa faible énergie de fuite. Toutefois, les eNVRAM présentent également une longue latence d'écriture et une endurance limitée. Pour at-

ténuer ces inconvénients, cette thèse présente SHyCache, une architecture de cache hybride et le modèle de programmation correspondant. SHyCache améliore les performances en allouant explicitement des variables avec des ratios élevés d'accès en lecture/écriture à la matrice eNVRAM, réduisant ainsi le temps d'accès, la consommation d'énergie et la surcharge de surface tout en maintenant une efficacité d'utilisation maximale et une facilité de programmation. Les tests de performance sur une gamme de variations de la hiérarchie de cache utilisant trois réseaux neuronaux, à savoir Inception v4, ResNet-50 et SqueezeNet 1.0, démontrent un espace de conception qui peut être exploité pour optimiser la performance, la consommation d'énergie ou l'endurance, tout en démontrant des gains de performance maximum de 1,7/1,4/1,3x et des réductions de consommation d'énergie de 5,1/5,2/5,4x.

Mots-clés : *Calcul en mémoire, calcul en cache, apprentissage automatique, intelligence artificielle, réseaux neuronaux, calcul approximatif, caches hybrides, SRAM, mémoires non volatiles émergentes, architecture des processeurs, informatique en périphérie*

Contents

Acknowledgements **i**

Abstract (English/Français) **v**

List of Figures **xv**

List of Tables **xix**

Acronyms **xxi**

1 Introduction **1**

1.1 Edge Devices and the von Neumann Bottleneck 2

1.1.1 Where to compute: NMP vs. iMC 3

1.1.1.1 Near Memory Processing (NMP) 4

1.1.1.2 in-Memory Computing (iMC) 4

1.1.2 What to compute: Simple vs. Arithmetic Operations . . 6

1.1.3 How to compute: iSC Architecture Challenges 7

1.2 Thesis Contribution 8

1.2.1 BLADE: A BitLine Accelerator for Devices on the Edge . 8

1.2.2 CAPPIEM: in-Memory Approximate Computation . . . 9

1.2.3 SHyCache: A Hybrid Cache Framework for NN Acceleration 9

2 BLADE: A BitLine Accelerator for Devices on the Edge **11**

2.1 Motivation for BLADE Design Choices 12

- 2.1.1 Accelerating via iMC 12
- 2.1.2 Implementing BLADE in the iSC Domain 13
- 2.1.3 Full-Stack Analysis Environment 14
- 2.2 Mitigating Data Corruption while Operating at High Frequency 16
  - 2.2.1 What are Local Bitlines? 17
  - 2.2.2 BLADE Methods of Operation 17
    - 2.2.2.1 Standard Read/Write Operations 18
    - 2.2.2.2 Standard iSC Operations 18
    - 2.2.2.3 LBL Enhanced iSC Operations 18
  - 2.2.3 LBL Enhanced iSC Cache Design Advantages 19
- 2.3 Logic to Support Arithmetic Operations 20
  - 2.3.1 Bitline Addition Architecture 22
  - 2.3.2 Improving Operation Throughput 25
    - 2.3.2.1 Manchester Carry Chain (MCC) 25
    - 2.3.2.2 Arithmetic Operation Pipelining 26
- 2.4 Electrical Validation and Design Space Exploration 27
  - 2.4.1 Functional Validation of BLADE 27
  - 2.4.2 Subarray Design Space Exploration 29
- 2.5 System-Level Integration and Functionality 31
  - 2.5.1 Operand Locality Constraints 34
  - 2.5.2 iSC Instruction Passing and ISA Support 37
    - 2.5.2.1 Memory-Mapped Implementation 37
    - 2.5.2.2 ISA Implementation 38
  - 2.5.3 Issuing ISC Operations to Memory 39
  - 2.5.4 Fetching and Allocating Operands 39
  - 2.5.5 Integration in the gem5-X Architecture Simulator 39
- 2.6 System-Level Benchmarking 40
  - 2.6.1 Edge Device Workloads 40
    - 2.6.1.1 Cryptography 40
    - 2.6.1.2 HEVC Video Processing 41
    - 2.6.1.3 Convolutional Neural Networks 41
  - 2.6.2 gem5-X Parameters 41
  - 2.6.3 McPAT Support 42
- 2.7 Benchmark Results 42
  - 2.7.1 Bitwise Operations/iSC Operation Count 43

2.7.2 FIR Filter/Cache Associativity 44
2.7.3 Convolution/Cache Capacity 45
2.7.4 Arithmetic Logic Optimization 46
2.8 Summary and Concluding Remarks 48

3 Accelerating iSC Operations via Approximate Computing 49
3.1 CAPPIEM: A Carryless Partial Product Inexact Multiplier 52
3.1.1 Selection of Reduction Operator 52
3.1.2 CAPPIEM Hardware Implementation 54
3.1.3 Implementation of CAPPIEM in BLADE 54
3.2 Fibonacci Code Word Quantization for NNs 57
3.2.1 Motivation 57
3.2.2 Countering Approximation Errors Via Fibonacci Code Word Quantization 57
3.2.3 Quantization Parameters for Fibonacci Code Quantization 58
3.3 Improving Accuracy through Incremental Network Quantization 60
3.3.1 Incremental Quantization Strategies 61
3.4 Experimental Setup 62
3.5 Experimental Results and Analysis 63
3.5.1 Hardware Synthesis and Analysis 63
3.5.2 Retraining Analysis 63
3.5.3 Comparison to the State-of-the-Art 66
3.6 Conclusion 68

4 Improving Capacity with Hybrid Caches 69
4.1 Related Work 72
4.1.1 Resistive Random Access Memory 72
4.1.2 Hybrid Cache Design and Allocation Strategies 72
4.1.3 Neural Networks 73
4.2 Hybrid Cache Architectural Design 73
4.3 Integrating SHyCache's Programming Model into Neural Network Frameworks 74
4.3.1 Enabling HC allocation at the Operating System Level . 74
4.3.2 Enabling HC allocation at the Application Level 75
4.3.3 Co-Implementing SHyCache with Other HC Allocation Strategies 76

4.4 Experimental Setup 76
4.4.1 gem5-X Simulator Parameters and Hybrid Cache Access Latency Simulation 77
4.4.2 Neural Network Benchmarks 78
4.5 Experimental Results and HC Optimization Analysis 78
4.5.1 Power Results 79
4.5.2 Performance Results 80
4.5.3 Endurance Results 82
4.5.4 Optimizing HCs for Performance, Power, or Endurance 83
4.6 Integration with BLADE 84
4.6.1 Pseudo-Local Group Integration 84
4.6.2 SHyCache/BLADE Operand Locality 85
4.7 Conclusion 85

5 Conclusion and Future Work 87
5.1 BLADE: A Base for Future iMC Exploration 87
5.2 Exploring Opportunities for Enhancing BLADE 89
5.3 Future Work 90
5.3.1 Validation of BLADE Fabrication 90
5.3.2 FPGA Implementation to Reduce Simulation Time . . . 91
5.3.3 BLADE Compiler Support 92
5.3.4 Further Refinement/Enhancement of BLADE Hardware and Architecture 92
5.4 List of Publications 94

Bibliography 95

Curriculum Vitae 107

List of Figures

1.1 Devices connected to the internet. Embedded IoT devices have seen a 10x increase in the last decade. 2
1.2 The various computing paradigms. The von Neumann architecture suffers from long access latency. Near Memory Processing (NMP) performs computation in tightly coupled logic near the memory array. in-Memory Computing (iMC) performs computation either directly in the memory or in the periphery immediately surrounding the array. 3
1.3 Cache subarray with `and/nor/xor` bitline computing on values A=0 and B=0. Bitwise operations are performed by first (a) precharging the bitlines, then (b) activating multiple wordlines, thus discharging the bitlines through the connected bitcells. . 6

2.1 BLADE fabricated on the Rosetta and Darkside chips in collaboration with the Integrated Systems Laboratory (IIS) at ETHZ. . 14
2.2 A variety of applications are utilized throughout this thesis to implement and benchmark BLADE from the circuit level through the architecture and kernel up to the application level. 15
2.3 (a) Bitcell shorting may cause one of the activated bitcells to flip. (b) avoids data corruption by lowering WL voltage, while (c) utilizes 8T bitcells to isolate bitcell contents. 16
2.4 LBL read ports prevent data corruption while performing iSC operations. 17

2.5 Transient simulations of bitwise operations @0.6V on (a) previously published architectures, and (b) the BLADE architecture. 19
2.6 Layout of a 256×2×32 BLADE memory with schematic diagram of the BLADE periphery. 21
2.7 Block schematic of the modified WL decoder used in this work. 22
2.8 4-bit shift-and-add multiplication as performed by BLADE. . . 24
2.9 MCC architecture with reduced transistor count resulting from simplified Generate and Propagate signal generation. 25
2.10 Timing diagrams of multiplication with (a) no optimization, (b) MCC, (c) pipelining, and (d) add-forward line. 26
2.11 Maximum frequency of bitwise operations vs. memory supply voltage at 28nm CMOS. 28
2.12 (a) Energy, area, and delay variations across cache geometries. (b) The normalized energy, area, and delay design space, with maximum values for each metric equal to 1. 30
2.13 a) BLADE implementation within the cache, represented by highlighted boxes. b) BLADE implementation under Local Group (LG) way multiplexers. 32
2.14 Instruction flow in BLADE cache from time of issue by processor to operation completion. 33
2.15 The operands in sets 0 and 1 have matching offsets and set LSBs, and differing set MSBs. Hence, they share BL logic, but not LGs. 35
2.16 a) iSC operand parameter format. b) Passing BLADE operands in ISA implementation mode utilizes both the address and data bus. 37
2.17 NEON/BLADE runtime/energy results on bitwise operations. . 43
2.18 Runtime/energy results for NEON/BLADE on FIR operations. . 44
2.19 Runtime/energy results for NEON/BLADE on convolution operations. 45
2.20 Performance trends for different levels of arithmetic pipelining. 47

3.1 Karnaugh maps for an exact half adder, as well as `or` and `xor` gates. While `xor` gates compute the sum value exactly in all instances, `or` gates provide the least error when the carry value is considered. 52

3.2 (a) 4-bit multiplication with carryless partial product summation enabled by `or` gates. (b) 8x8 adder array with FA gates replaced by `or` gates to reduce area and power consumption. The ratio of replaced adders approaches a limit of 50% as bit width increases. 53
3.3 4-bit shift-and-add multiplication as performed by BLADE. . . 54
3.4 Embedded shift logic with CAPPIEM extension in green. Extension performs an `or` operation between the current and previous bit of the operand. 55
3.5 Embedded shift logic with CAPPIEM extension in green. Extension performs an `or` operation between the current and previous bit of the operand. 56
3.6 Naive carryless partial product multiplication will lead to errors and reduced accuracy (a). Such errors can be avoided by quantizing NN weights according to Fibonacci code words (b). 57
3.7 Three incremental quantization strategies for Fibonacci incremental retraining; (a) Random, (b) Proximal, and (c) Distant. In each retraining step, a fraction of weight values are quantized to the nearest Fibonacci code word and frozen, represented by green boxes. 59
3.8 Placed and routed 32 bit CAPPIEM in 28nm CMOS. 62
3.9 (a) Area and Power-Delay-Product (PDP) reduction comparison between this work against other AMs for 8x8 multiplier architectures. (b) Mean Relative Error Distance (MRED) of this work in comparison to other AMs. This work presents high area and PDP reductions in relation to its low MRED. 64
3.10 DenseNet-121 accuracy during the quantization/retraining process utilizing the distant retraining strategy. Retraining enables to recoup nearly all accuracy loss. Weights are quantized conservatively, as described in Section 3.3.1. The blue line represents the percentage of weights quantized at each step. 65

4.1 Read/write access ratios in relation to total read accesses. Weight access accounts for nearly 40% of reads. 70

4.2 SHyCache is a HW/SW stack (a) that enables efficient use of a hybrid cache (b). 72
4.3 Power consumption of all-SRAM, SRAM+eNVRAM, and all-eNVRAM caches for Inception (I), ResNet (R) and SqueezeNet (S) Neural Networks (NNs). 79
4.4 caption 81
4.5 STT-MRAM bitcell flips across varying L1 (a) and Level 2 (L2) (b) cache sizes. 83
4.6 a) SHyCache + BLADE architecture with eNVRAM output routed to SRAM subarray. 84

5.1 Chips fabricated in collaboration with the Integrated Systems Laboratory (IIS) at Swiss Federal Institute of Technology in Zürich (ETHZ). 91

List of Tables

2.1 Worst Case iSC Energy/Frequency Values in a 256×64 array with 2 Local Groups (LGs). 29
2.2 Simulator Parameters 42
2.3 Cycles for 8/32-bit multiplication at different pipeline levels @2Ghz 47

3.1 Cumulative Fraction of Weights to Quantize at Each Incremental Quantize-and-Retrain Step 61
3.2 Accuracy Degradation for quantization strategies after INQ . . 66
3.3 Comparison of this work to other approximate multiplier NNs 67

4.1 Simulator Parameters 77
4.2 Neural Network Benchmark Parameters 78

Acronyms

$\overline{\mathbf{BL}}$ BitLine Bar. 7

$\overline{\mathbf{GBL}}$ Global BitLine Bar. 18, 23

6T 6 Transistor. 8, 11, 16–18, 47, 48, 72, 73

ACL Arm Compute Library. 15, 41, 45, 76, 78

ADC Analog-to-Digital Converter. 4, 5, 13

AM Approximate Multiplier. 50–52, 56, 57, 60, 62, 63, 65–68

ASIC Application Specific Integrated Circuit. 12

BL BitLine. 5–7, 11, 15, 16, 19, 20, 22–24, 26, 27, 29, 31, 32, 34, 47

BLADE BitLine Accelerator for Devices on the Edge. iii, iv, xii, xiii, 2, 8–15, 17–25, 27, 28, 31–52, 54, 55, 68–71, 84, 85, 87–94

CAPPIEM CArryless Partial Product InExact Multiplier. iv, xiii, 9, 50–52, 54–57, 62, 63, 67, 68, 89

CNN Convolutional Neural Network. 41, 45, 46

CPU Central Processing Unit. iii, 2, 3, 5, 8, 9, 12, 31–34, 38–40, 42, 44, 45, 48, 77, 88, 93

DAC Digital-to-Analog Converter. 20

DMA Direct Memory Access. 39, 93

DRAM Dynamic Random Access Memory. iii, 2, 4, 5, 13, 36

eNVRAM emerging Non-Volatile Memory. iii, iv, 3, 4, 11, 13, 70–78, 80, 82, 84, 85, 90

ETHZ Swiss Federal Institute of Technology in Zürich. 90

FA Full Adder. 52, 54, 63, 65

FCQ Fibonacci Codeword Quantization. 50, 51, 58–63, 65, 66, 68

GBL Global BitLine. 17, 18, 29, 32, 55, 84

gemmlowp low-precision general matrix multiplication. 58

GPU Graphics Processing Unit. 2, 4, 50, 57, 62, 71

HC Hybrid Cache. iv, 70–72, 74, 76–85

HEVC High Efficiency Video Encoding. 41

HMC Hybrid Memory Cube. 4

IIS Integrated Systems Laboratory. 14, 90, 91

iMC in-Memory Computing. iii, 4–6, 8, 12, 13, 70, 87, 92

INQ Incremental Network Quantization. 50, 51, 59–63

IoT Internet of Things. 1, 40

ISA Instruction Set Architecture. 31, 37, 38, 42, 77, 88

iSC in-SRAM Computing. iii, iv, xii, xv, 5–8, 11, 13, 16–20, 23, 24, 26–29, 31, 34–39, 41, 42, 45–48, 71, 88

L1 Level 1. 33, 42, 44–46, 74, 77, 79, 80, 82–84, 88, 93

L2 Level 2. xiv, 33, 42, 74, 77, 79, 80, 82–84

LBL Local BitLine. 17–20, 31, 35, 36

LG Local Group. xii, xv, 17, 18, 27–29, 31, 32, 34, 48, 55, 56, 68, 92

LSB Least Significant Bit. 24, 34–36, 49, 63, 82, 85

LVT Low Voltage Threshold. 27

MCC Manchester Carry Chain. 25, 26, 28, 29

MRED Mean Relative Error Distance. 52, 62, 63

MSB Most Significant Bit. 23, 24, 34–36, 49, 55, 56, 75, 85

NMP Near Memory Processing. 4, 5

NN Neural Network. xiv, xv, 5, 7, 9, 10, 50, 51, 57, 58, 62, 66–71, 73, 74, 76, 78–80, 82, 83, 85, 90, 93

PCM Phase Change Memory. 3, 4

PDP Power-Delay-Product. 62, 63, 68

PLG Pseudo-Local Group. 84, 85, 92

PnR Place and Route. 15

RVT Regular Voltage Threshold. 27

SHA3 Secure Hash Algorithm 3. 41, 43

SHyCache HW/SW Stack for Hybrid Caches. iv, 10, 71, 73–80, 84, 85, 90

SIMD Single Instruction Multiple Data. iii, 3, 5, 8, 13, 41, 42, 48, 49, 84, 85

SotA State-of-the-Art. 4, 8, 50, 67, 76, 87, 88

SRAM Static Random Access Memory. iii, iv, 3, 5, 6, 10, 11, 13, 14, 16, 17, 20, 22, 27, 31, 32, 36, 47, 70–74, 76–82, 84, 85, 87, 90, 94

STT-MRAM Spin-Transfer Torque Magnetoresistive Random Access Memory. 3, 4, 77, 79, 80, 82, 83

WL WordLine. xi, xii, 5, 7, 16–18, 20, 22, 23, 27, 29, 31, 34–36, 42, 48

Introduction

EMBEDDED systems have seen explosive growth in the last decade. At a market value of $86.5 billion and expected to grow to over $125 billion by 2025 [1], the embedded systems market has been driven by multiple factors, such as the rise of autonomous driving [1] and the internet of things [2]. For the first time in 2020, there are more Internet of Things (IoT) devices connected to the internet than non-IoT devices, as illustrated in Figure 1.1, and this trend is only expected to continue [3]. The span of fields utilizing edge devices is ubiquitous, ranging from industrial control to transportation to medical [2]. As use cases increase, embedded systems are being tasked with increasingly complex applications, such as computer vision, speech recognition, seizure detection, and water and soil analysis, just to name a few. As their ubiquity increases, embedded devices are also becoming smaller and consuming less energy. This, coupled with the high cost of wireless transmission and exponential increases in sensor networks and the consequent "data explosion" [4] has led to the paradigm known as "edge computing", where data is preprocessed at the embedded level before being sent upstream to the cloud. This conflict of interest between enhanced computing power and reduced energy consumption has spurred research into innovative technologies that enhance embedded device capabilities while maintaining a low power budget. This thesis proposes and develops three such lines of research, namely, in-memory computing, hybrid memory systems, and inexact arithmetic units. All three fields extend from a novel in-

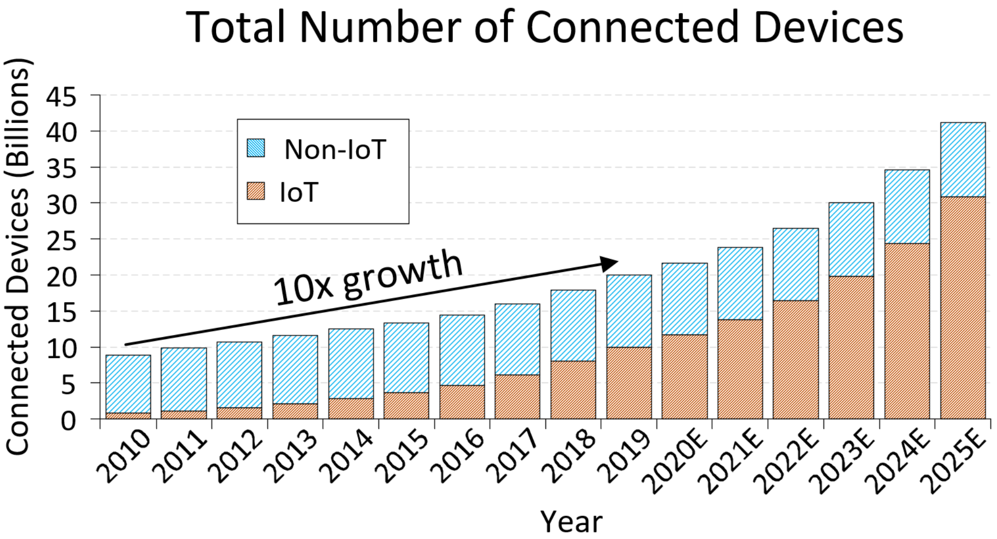


Figure 1.1: Devices connected to the internet. Embedded IoT devices have seen a 10x increase in the last decade.

memory computing architecture designed by the author and coined BLADE, or a BitLine Accelerator for Devices on the Edge, which will be the centerpoint of this work.

1.1 Edge Devices and the von Neumann Bottleneck

As edge devices become more ubiquitous, they are being increasingly utilized to perform compute intensive tasks traditionally reserved for servers with access to standard accelerators such as Graphics Processing Units (GPUs) [5, 6]. The need to improve performance while maintaining low area and energy overhead presents a unique challenge in embedded device research. This is further complicated by the challenge faced by embedded and server devices alike known as the von Neumann bottleneck [7], where limited I/O bandwidth between the compute and storage elements in an architecture further limit performance. To alleviate this bottleneck and improve embedded device performance while maintaining low energy and area overhead, a plethora of so-called smart memory device and strategies [8] have been proposed. These devices function by computing at the point of data storage instead of moving data to the Central Processing Unit (CPU). Such architectures

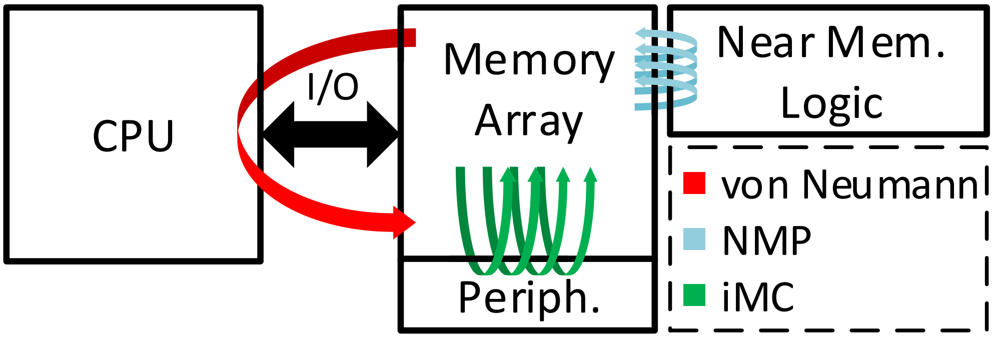


Figure 1.2: The various computing paradigms. The von Neumann architecture suffers from long access latency. Near Memory Processing (NMP) performs computation in tightly coupled logic near the memory array. in-Memory Computing (iMC) performs computation either directly in the memory or in the periphery immediately surrounding the array.

are implemented alongside a variety of storage devices, including Dynamic Random Access Memory (DRAM), Static Random Access Memory (SRAM), and emerging Non-Volatile Memory (eNVRAM) devices such as Spin-Transfer Torque Magnetoresistive Random Access Memory (STT-MRAM) or Phase Change Memory (PCM). All smart memories share two inherent benefits, namely, they reduce memory movement, and secondly, they enable massive Single Instruction Multiple Data (SIMD) operations by computing on full rows of data simultaneously.

When making design decisions for smart memories, balancing many competing factors is necessary, including functionality and generalizability, integration and implementation details, and area and energy costs. Specifically, three factors must be asked when considering the benefits and trade-offs of such architectures, namely, where to compute, what type of computation to perform, and what challenges arise in performing said computation.

1.1.1 Where to compute: NMP vs. iMC

Smart memories may be roughly broken into two categories, Near Memory Processing and in-Memory Computing. Figure 1.2 illustrates the locations of these compute paradigms relative to the CPU and memory array.

1.1.1.1 Near Memory Processing (NMP)

Near Memory Processing (NMP) is the practice of placing compute logic near memory in an effort to reduce data movement latency between said logic and memory [9]. Many proposed NMP architectures allocate compute blocks on the logic layer of Hybrid Memory Cube (HMC) DRAM [10–13]. Other works couple GPU architectures with 3D stacked memories [14, 15]. Still others utilize reconfigurable logic near the DRAM [16–18].

While NMP computing shows promise, there is still much research to be done to validate its feasibility. First, from an integration standpoint, NMP poses a challenge in regards to virtual-physical memory translation and managing cache coherency [9]. Second, from an application viewpoint, few works provide potential solutions for optimizing algorithms to utilize NMP units while accounting for data locality [15, 19]. Finally, the HMC logic layer area/power budget is very constrained, thus limiting NMP logic complexity [11].

1.1.1.2 in-Memory Computing (iMC)

The definition of in-Memory Computing (iMC) is fuzzier than NMP. At its core, it refers to computation done physically within the memory array. The ambiguity in the term iMC arises when considering the amount of periphery utilized to perform the computation. Some amount of periphery is always necessary to interpret the array output, whether that be an Analog-to-Digital Converter (ADC) for analog iMC devices, or logic gates for digital devices. How much periphery is acceptable while still classifying a device as iMC is debatable. The iMC device described in this work results in 8% periphery area overhead, well under the overhead many iMC State-of-the-Art (SotA) works incur, and therefore the term iMC is ascribed to this device.

In-memory computation can be accomplished in a variety of ways depending on the memory technology. In eNVRAM devices such as STT-MRAM and PCM, by far the most popular strategy to perform computation is the usage of crossbar arrays composed of the variably resistive memory elements [20–22]. By injecting input values into this array, Kirchhoff's circuit laws can be used to multiply input values by values stored in the memory elements before summing them on a single line and converting them into digital form via

ADC. While this strategy has the advantage of potentially performing a huge number of operations simultaneously, it has faced several long-standing challenges In particular, device non-ideality resulting in erroneous results leads to the necessity of area and energy hungry ADC.

On the other hand iMC performed in DRAM utilizes charge sharing between bitcells by activating multiple rows simultaneously and sensing the relative gain or drop in voltage on the bitline to perform majority functions [23]. Alongside the challenges of NMP introduced in the above section on NMP, such a strategy introduces many further difficulties in regards to data placement within the DRAM and the destructive nature of charge sharing. Thus, the data must be copied to dedicated compute rows, mitigating the energy and latency performance gains [23].

A final form of iMC is in-SRAM Computing (iSC). iSC architectures allocate compute logic immediately adjacent to SRAM bitcell arrays, either in dedicated iSC accelerators or in the preexisting cache of the memory hierarchy. iSC architectures take advantage of the SRAM array's BitLine/WordLine (BL/WL) structure to perform massive SIMD computations at a small area/energy cost [24]. Many iSC architectures have been proposed [25–27], with each demonstrating different aspects of the technology. These architectures exploit the inherent data locality found in applications such as Neural Networks (NNs) to perform these operations, similar to other SIMD accelerators such as ARM's NEON [28] and Intel's AVX [29] architectures. Further, when integrated into a cache hierarchy, iSC architectures reduce energy consumption by (a) taking advantage of the cache's set/way allocation scheme to align data and avoid unnecessary data movement, and, (b) reducing data movement between the cache and the CPU. This is accomplished at a minimal area overhead, as the iSC architecture augments preexisting hardware. However, previous works suffer from various shortcomings in relation to simulation methodology, electrical design, or application support. In particular, many of the works lack thorough analysis of system-level integration implications, as well as demonstration of functionality within a full software stack. Also, data corruption within the SRAM array is a prominent problem for iSC architectures [24, 30], and the solutions proposed so far within literature either greatly reduce operating frequency [31] or area efficiency [25].

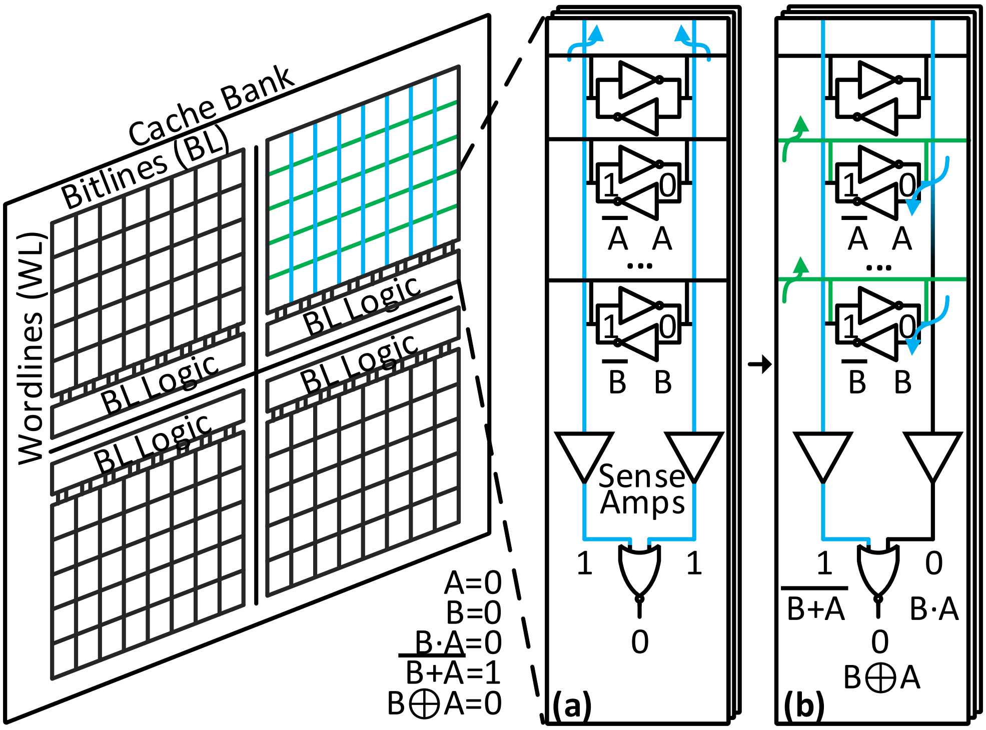


Figure 1.3: Cache subarray with `and`/`nor`/`xor` bitline computing on values A=0 and B=0. Bitwise operations are performed by first (a) precharging the bitlines, then (b) activating multiple wordlines, thus discharging the bitlines through the connected bitcells.

The devices described in this thesis are implemented in the domain of iSC. The author prefers this form of iMC for various reasons, including the maturity of SRAM technology, the digital nature of the computation, the straightforward implementation within the architecture, and easy integration with the kernel and application. Therefore, the focus of the remainder of this thesis will reside within the iSC domain.

1.1.2 What to compute: Simple vs. Arithmetic Operations

Choosing which operations to support in iSC architectures is not trivial, as there is a complex interrelationship between operation complexity, latency, throughput, and area overhead. Many iSC architectures support only bitwise operations through a technique known as BL computing, first demonstrated by Jeloka *et al.* [31], and illustrated in Figure 1.3. BL computing operates similarly to a standard cache read operation. A read operation involves precharging a pair of BLs for each bitcell to be read (Figure 1.3-a), and then activating a single WL to connect the row of bitcells to the BLs, resulting in BL discharge according to the row's contents. In contrast, BL computing involves the simultaneous activation of multiple WLs (Figure 1.3-b), with the resulting BL discharge computing two bitwise operations between the bits, with the BL and BitLine Bar ($\overline{\text{BL}}$) producing `and` and `nor` operations, respectively. Aga *et al.* extend this work by `nor`ing the two BLs, resulting in a `xor` operation [24]. This architecture forms the basis of many iSC publications [25, 32–36].

Conversely, multiple works support more complex workloads by implementing BL logic that performs arithmetic operations. Analog solutions such as those presented in [26, 37–40] modulate BL/WL voltages to perform arithmetic operations and utilize analog circuitry implemented under the subarray to sense and convert BL voltages to approximate digital results. On the other hand, solutions such as [27, 41, 42], as well as the accelerator presented in this work, utilize digital logic to compute exact solutions to arithmetic operations.

1.1.3 How to compute: iSC Architecture Challenges

The above iSC architectures have been successfully applied to a wide range of applications including query processing and in-memory checkpointing [24], cryptography [24, 32, 43], NNs [33, 37–40, 42, 43], finite state automaton [25], and video encoding [43], thus demonstrating their performance and energy benefits. However, iSC architectures face a variety of challenges at the system, architectural, and electrical level that must be overcome for such architectures to become widely adopted. These challenges include:

- Guaranteeing correct cache functionality at the electrical level.
- Supporting arithmetic operations such as addition and multiplication.
- Integration into the memory hierarchy while addressing issues such as memory translation, coherency, etc.
- Accurate benchmarking on a full software stack.

In the state-of-the-art literature, no single architecture succeeds in overcoming all of the aforementioned challenges. This thesis's contributions address each of these challenges in detail.

1.2 Thesis Contribution

In this thesis, I present contributions to the aforementioned iMC/NMP challenges. Namely, I present a BitLine Accelerator for Devices on the Edge (BLADE), an iSC accelerator that contains novel innovations allowing it to perform arithmetic computations and run at higher operating frequencies than previously reported in SotA works. I develop this architecture across the entire HW/SW stack, allowing for further modifications and enhancements from the hardware and architecture up to the kernel and application level. With this flexible platform in place, I proceed to propose various further modifications and enhancements that reduce operation latency, improve capacity, and improve application performance. Lastly, I provide insight on future work for continued exploration in the iSC space.

1.2.1 BLADE: A BitLine Accelerator for Devices on the Edge

Chapter 2 presents BLADE, the primary contribution of this thesis. BLADE is an iSC architecture targeted specifically for implementation in low-power edge devices. It performs massive SIMD bitwise and arithmetic computations required by emerging edge device workloads directly in-memory, obviating the need for costly data movement or time-consuming CPU computation. BLADE addresses each of the aforementioned shortcomings prevalent in other iSC architectures. Design choices are motivated from the transistor, architectural, and system levels in order to demonstrate the architecture's energy and performance characteristics at all levels of abstraction. BLADE divides wordlines into isolated subgroups called local groups, eliminating the risk of data corruption, and utilizes a carry lookahead adder and operation pipelining to improve iSC operating frequency by 2.3x-2.8x compared to previous iSC architectures. BLADE maintains a low (8%) area overhead and functions at the lowest operating voltage (0.6V) of any 6 Transistor (6T) bitcell iSC architecture. It is integrated into an edge device cache hierarchy and benchmarked within a fully functioning Linux environment, enabling consideration of system-level events such as cache misses, coherence requests, and CPU/cache hierarchy pipeline stalls. BLADE is validated in 28nm CMOS technology and benchmarked on the gem5-X architectural simulator using three edge device workloads, namely, cryptography, image processing, and NNs, demonstrating 4x/6x, 6x/2x, and 3x/1.5x performance/energy gains, respectively.

Using BLADE as a base, many further strategies are possible for enhancing its in-memory computing capabilities. The next chapters of this thesis implement and analyze two such possibilities, namely, hybrid caches, and approximate computing.

1.2.2 CAPPIEM: in-Memory Approximate Computation

One of the major drawbacks of BLADE's low area footprint is the high latency of multiplication, the core operation of NNs. In an effort to reduce multiplication latency, in Chapter 3 I introduce a CArryless Partial Product InExact Multiplier (CAPPIEM), that can be implemented in BLADE at a small area overhead, reducing multiplication latency by 2x. In order to mitigate accuracy loss due to approximated products during inference, I take advantage of a unique property of CAPPIEM, namely, if one of the operands has no consecutive ones in its binary representation, the product will be exact. By iteratively quantizing weights to such values, CAPPIEM acts as an exact multiplier. This quantization strategy accelerates retraining by up to 300x while reducing accuracy by <2% for three benchmarks. I also implement CAPPIEM as a standalone approximate multiplier, reducing area footprint and power-delay-product by 73/43%, respectively.

1.2.3 SHyCache: A Hybrid Cache Framework for NN Acceleration

Since Alexnet in 2012, the depth of NNs has exploded, reaching for example 1000 layers in the ResNet series [44]. Recent years have seen smaller networks coming back into fashion; however, the memory footprint of "small" NNs often still measure in the order of MBs [45]. As such, larger capacity memory arrays can help to improve runtime by reducing data movement. In this vein, Chapter 4 introduces a HW/SW Stack for Hybrid Caches (SHyCache), a non-volatile/SRAM hybrid cache that increases the size of the cache available to the processor without increasing area overhead. A deterministic data allocation strategy that takes advantage of the static nature of NNs' weights is also introduced, along with a C++ support library that can be integrated into NN frameworks such as Pytorch or Tensorflow. I performed a design space exploration over a wide range of cache hierarchies and capacities to characterize SHyCache. Benchmarks of SHyCache with three NNs demonstrate 1.4/1.3/1.7x performance gain and 5.1/5.2/5.4x power consumption reduction. I finally provide suggestions for integration with BLADE in the Future Work section.

2

BLADE: A BitLine Accelerator for Devices on the Edge

As discussed in Chapter 1, BLADE is an iSC accelerator designed specifically for low power edge devices. The primary motivation in developing BLADE was to create an iSC accelerator that could be implemented in mature technology available on the market today. Hence, BLADE is implemented in SRAM, as opposed to the emerging eNVRAMs currently being developed. Development of a full hardware/software stack for BLADE accounts for each level of abstraction from the transistor level design up to application design. BLADE's compute architecture utilizes BL computing as described in Section 1.1.2, with further developments to run at high frequency and provide enhanced compute capability.

A summary of this chapter's contributions is presented as follows:

- BLADE is introduced. BLADE is a holistically designed and simulated iSC architecture capable of arithmetic operations, designed specifically for edge devices.
- BLADE utilizes local bitlines, operation pipelining, and carry-lookahead addition to achieve the best voltage/frequency Pareto curve of any 6T iSC architecture (0.6V/416MHz-1V/2.2GHz for bitwise operations). Its electrical design is validated in 28nm CMOS technology.
- BLADE is situated within the cache hierarchy. Its controller functionality and application level interface is detailed.

- BLADE is benchmarked on the gem5-X architectural simulator via three edge device workloads, namely, cryptography, image processing, and neural networks, demonstrating 4x/6x, 6x/2x, and 3x/1.5x performance/energy gains respectively.

The remainder of the chapter is organized as follows. Section 2.1 provides motivation for the high-level design choices made when designing and implementing BLADE. Section 2.2 details the subarray optimizations allowing BLADE to run at high frequency with a low energy consumption while avoiding data corruption. Section 2.3 explains how arithmetic operations are supported. Section 2.4 provides information on BLADE's electrical validation as well as subarray designn space exploration results. Section 2.5 details BLADE's integration in the cache hierarchy and its interaction with the rest of the architecture. Section 2.6 details the methodology for evaluating BLADE at the system level. Section 2.7 illustrates benchmark results. Finally, Section 2.8 provides concluding remarks.

2.1 Motivation for BLADE Design Choices

The concept of BLADE was born as an answer to the question, "What can be done to accelerate edge applications in a realistic, easily implementable manner, while mitigating area and energy overhead?" Answering this question led to a series of design choices that defined the parameters of BLADE. This section seeks to provide a high-level overview of these choices before delving into the specifics of the design.

2.1.1 Accelerating via iMC

In an effort to optimize edge devices, a wide variety of research is being performed into a range of technologies, such as systolic arrays [46], Application Specific Integrated Circuits (ASICs) [47], and simplified CPU pipelines [48]. A challenge facing all the aforementioned research avenues is the need to move data between memory and compute devices. On the other hand, as discussed in Chapter 1, iMC differentiates itself from other strategies by performing compute directly in-memory fabric. Such an architecture provides multiple benefits.

Namely, iMC:

- reduces data movement between memory and compute elements, along with the associated latency and energy consumption.
- can reuse existing architecture elements, thus reducing area overhead.
- enables SIMD operations in-memory, capable of performing 100s-1000s of simultaneous operations.

For these reasons, I chose to pursue an iMC solution in answering the above question. Once this broad domain space was established, narrowing down the type of iMC technology was addressed.

2.1.2 Implementing BLADE in the iSC Domain

As stated in Section 1.1.1.2, in-Memory Computing (iMC) accelerators can be implemented alongside a variety of memory devices. When considering the different options, namely SRAM, DRAM, and eNVRAMs, SRAM was chosen. iSC has several benefits:

- SRAM is a mature technology, having been in use since the 1960's [49]. Its well understood physical properties are widely supported via simulation tools and fabrication companies. SRAM designs are less complicated to implement and characterize, simplifying integration into existing commodity devices.
- SRAM is commonly found in architectures ranging from edge devices up to server systems. This enables BLADE integration at all levels of system complexity, Also, it enables reuse of existing components, greatly reducing its area overhead in comparison to a dedicated accelerator.
- iSC most commonly utilizes digital logic circuits as opposed to analog. While this reduces the number of possible operations per cycle, the lack of need for ADC greatly reduces area overhead. Digital computation also produces exact operation results, obviating the need to consider the impact of device utilization on application accuracy.

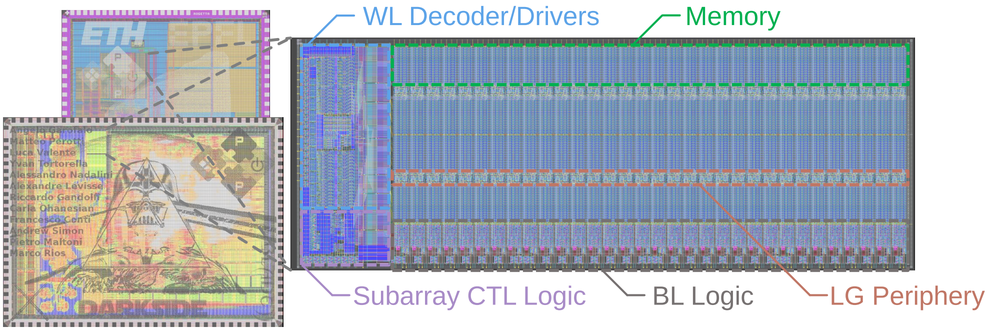


Figure 2.1: BLADE fabricated on the Rosetta and Darkside chips in collaboration with the Integrated Systems Laboratory (IIS) at ETHZ.

Beyond positioning BLADE in the SRAM domain, it is further targeted primarily at the cache hierarchy. The reasons for this design decision are explained in Section 2.5. It should be noted, however, that BLADE can indeed be implemented alongside any SRAM subarray. This is demonstrated by the two fabrication projects performed in collaboration with the Integrated Systems Laboratory (IIS), illustrated in Figure 2.1, in which BLADE is implemented in the chip's tightly coupled scratchpad memory. Such an architecture has the benefits of reduced area overhead due to the absence of tag arrays and more control over data placement, in exchange for increased application complexity. These fabrications will be further discussed in Chapter 5.

2.1.3 Full-Stack Analysis Environment

To make the most compelling argument possible for the utility of BLADE, I decided early in the design process to address every level of the HW/SW stack, from the hardware and circuit design level up to the application level. This was to avoid any possible oversights that could occur by making assumptions about implementation at different levels in the stack, such as overly optimistic hardware or architectural parameters, or simplistic benchmarks unrepresentative of real-world applications.

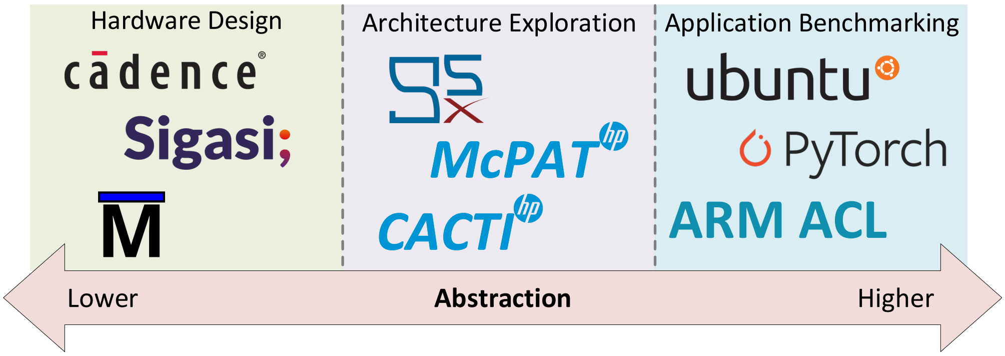


Figure 2.2: A variety of applications are utilized throughout this thesis to implement and benchmark BLADE from the circuit level through the architecture and kernel up to the application level.

Developing along the full stack requires a range of software to accomplish, as illustrated in Figure 2.2:

- At the hardware level, Cadence Virtuoso and Innovus [50, 51] are used to layout and Place and Route (PnR) full-custom designs. Sigasi [52] is used to code semi-custom designs, and Modelsim [53] is used to simulate all designs in pre-and post-PnR.
- At the architecture level, the architectural simulator gem5-X [54] is used to integrate BLADE into a full-system architecture. Hewlett-Packard's McPAT [55] and Cacti [56] are used in conjunction with gem5-X activity traces to calculate energy consumption during application runtime. Energy values collected in hardware simulations are also used for this analysis. Further information is provided in Section 2.5.5.
- At the application level, all applications are benchmarked in a modern version of the Ubuntu Linux distro [57]. Applications are developed in multiple frameworks, including PyTorch [58], the Arm Compute Library (ACL) [59], and other sources. These benchmarks will be described in more detail in the relevant sections.

Beginning from the base concept of BL computing described in Section 1.1.2, the remainder of this chapter describes BLADE in increasing levels of abstraction, from the hardware implementation up to application development.

2.2 Mitigating Data Corruption while Operating at High Frequency

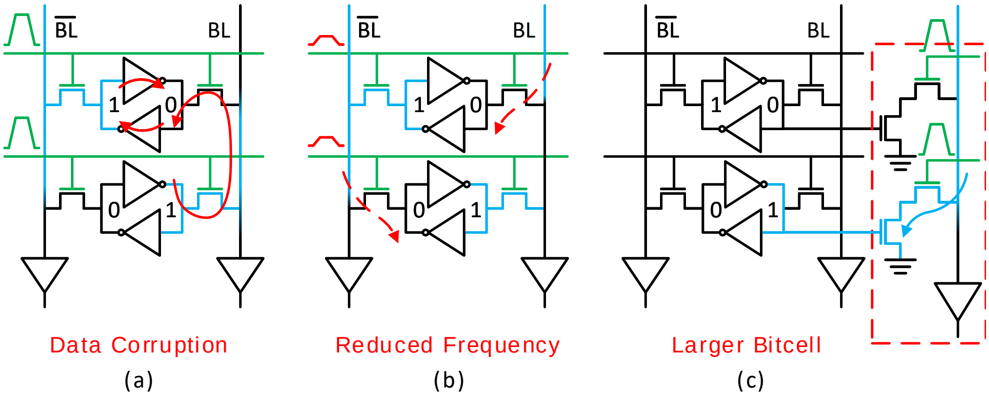


Figure 2.3: (a) Bitcell shorting may cause one of the activated bitcells to flip. (b) avoids data corruption by lowering WL voltage, while (c) utilizes 8T bitcells to isolate bitcell contents.

One of the biggest challenges for iSC architecture's based on BL computing as described in Section 1.1.2 is the avoidance of data corruption. When activating multiple WLs in a conventional 6T SRAM array, as detailed in Section 1.1.2, a short is produced between the activated bits. It is possible that, due to process variation during fabrication, the runtime content of the cells, or transistor aging and degradation, the contents of one bit can cause another bit to flip, as illustrated in Figure 2.3-a. Preventing data corruption has been achieved in previous literature via multiple methods. The first is to aggressively limit WL voltage, as done in [24,31,38,41] and illustrated in Figure 2.3-b. However, such a technique greatly reduces operating frequency, reported at 800MHz@1V [31] and 475MHz@1.1V [42]. Another method, demonstrated in Figure 2.3-c, is the introduction of 8T or larger bitcells, which isolate the bitcell's contents from the BLs [25,27,32,40], at the cost of a significantly less area efficient SRAM subarray as 8T bitcells are up to 30% larger over 6T [60]. A third method is to use pulse width modulated WLs such that no two WLs are active simultaneously [37], removing the danger of data corruption, but at the cost of a 2.35x increase in periphery area. Finally, nonconventional technologies can be used, such as monolithic 3D integration [34, 35], or deeply depleted channel technology [61]. Emerging technologies present their own challenges however; for example, deeply depleted channel technology demonstrates stability issues [62] and disturb risks, and results in poor performance/voltage scaling, limiting the maximum clock frequency to 100Mhz@0.6V.

In contrast to previous literature, BLADE utilizes local bitlines to maintain a high operating frequency and low area overhead, avoid data corruption, and facilitate simple implementation in conventional 6T SRAM arrays.

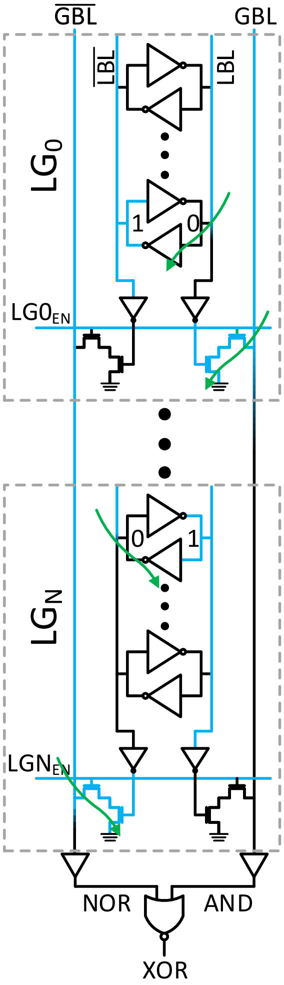


Figure 2.4: LBL read ports prevent data corruption while performing iSC operations.

2.2.1 What are Local Bitlines?

Local BitLines (LBLs), illustrated in Figure 2.4, are a cache optimization present in many architectures. LBLs divide groups of WLs into Local Groups (LGs), where all WLs in an LG share an LBL pair, which is connected to the Global BitLine (GBL) pair via I/O circuitry. LBLs reduce parasitic capacitance resulting from excessively long GBLs, thus improving cache energy efficiency and reducing delay. LBLs also improve static noise margin by isolating bitcells into small groups, reducing leakage interference. For these reasons, LBLs are already implemented in caches [63].

2.2.2 BLADE Methods of Operation

BLADE re-utilizes LBLs, taking advantage of the isolation provided to perform high frequency iSC operations reliably. A BLADE enhanced cache can perform three sets of memory operations, namely, standard read/write, slow same-LBL iSC operations, and fast multi-LBL iSC operations.

2.2.2.1 Standard Read/Write Operations

Accessing a single WL of the cache performs a standard read or write operation. In order to perform a read operation, the GBLs and the LBLs of the LG to which the target set belongs are first precharged to V_{dd}. Then, when the WL is activated, one of the LBLs discharges. This discharge is propagated through the local read port to the GBLs (illustrated in Figure 2.4, where the discharge is sensed by one of the two single-ended sense amplifiers attached to the GBLs. The addition of local read ports in fact transforms the standard 6T bitcells into pseudo two-port bitcells. Therefore, simultaneous read + write operations are possible if the accessed words are located in different LBLs.

2.2.2.2 Standard iSC Operations

BLADE can also perform iSC operations in the manner described in Section 1.1.2. However, when performing operations between bitcells in opposing states, there is a risk of flipping a bitcell if the PMOS transistor of one bitcell is weaker than the access and pulldown transistors of the other bitcell, as shown in Figure 2.3-a. To counteract this problem, it is necessary to greatly reduce the WL voltages, resulting in a low operating frequency (<1GHz) [24, 31].

2.2.2.3 LBL Enhanced iSC Operations

In order to avoid the aforementioned bitcell flipping while maintaining high operating frequency, BLADE reuses LBLs and their resulting LGs. When WLs belonging to different LGs are simultaneously activated, the local read ports isolate bitcells from each other, eliminating the risk of bit flipping, as demonstrated in Figure 2.4. Activating each WL results in a simple read operation within the local group, which is propagated to the GBLs by local read ports, resulting in an iSC operation between LGs, where the GBL represents an `and` operation, while the Global BitLine Bar ($\overline{\text{GBL}}$) represents a `nor`. LGs allow BLADE to maintain a high operating frequency without a reduction in WL voltage, while still utilizing small 6T bitcells. It should be noted that introducing LBLs induces a data placement constraint, namely, that operands cannot occupy the same LG, as discussed in Section 2.5.1.

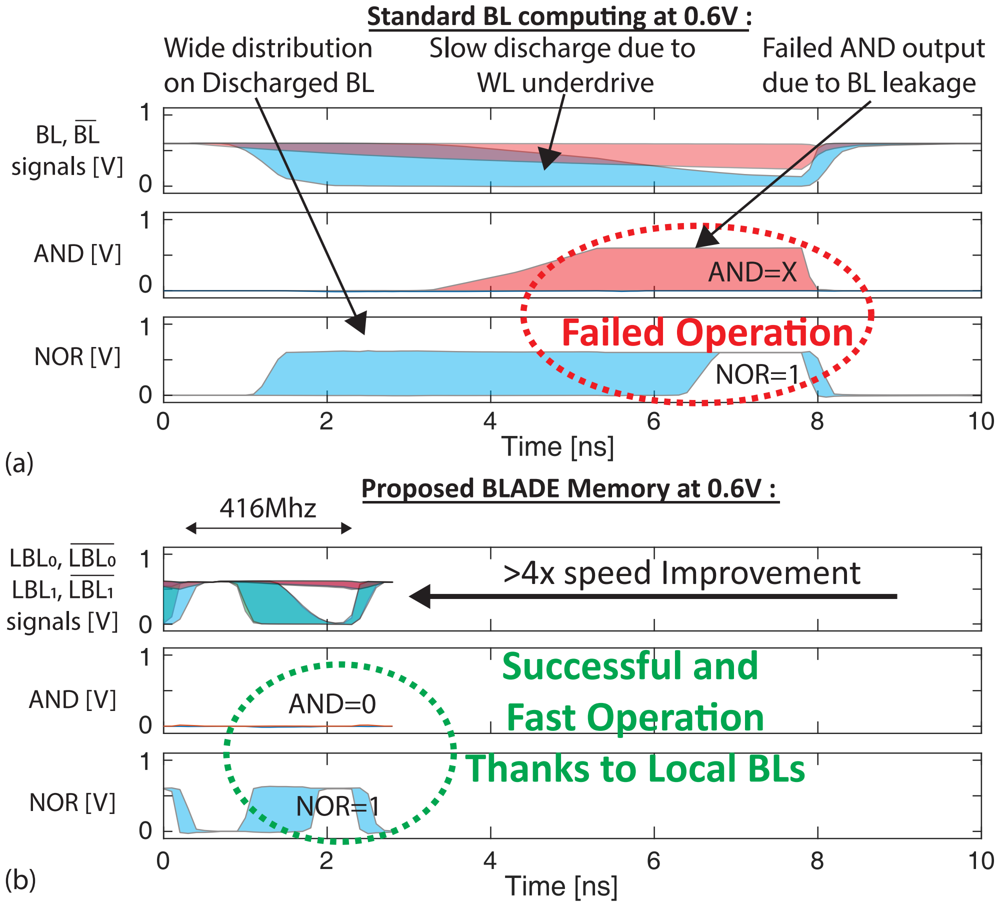


Figure 2.5: Transient simulations of bitwise operations @0.6V on (a) previously published architectures, and (b) the BLADE architecture.

LBLs also enable iSC operation in low voltage environments. Figure 2.5 illustrates the difference in functionality between (a) standard iSC operations and (b) LBL enhanced operations at a V_{dd} of 0.6V. LBL enhanced operations complete 4x faster than standard operations, and in fact the and operation fails to converge, due to excessive BL leakage.

2.2.3 LBL Enhanced iSC Cache Design Advantages

Utilizing LBLs to enable iSC operations provides multiple advantages over state-of-the-art iSC architectures. As discussed in Section 2.2, most iSC architectures require a significant redesign of either the bitcell array or periphery to enable computation. BLADE, on the other hand, introduces minimal changes to the cache architecture. The LBL design, as well as the use of digital computation in contrast to analog, allow BLADE to function over a large voltage range, thus easing implementation across a wide range of cache architectures and making BLADE suitable for low power edge devices. Ease of implementation is further facilitated by the fact that no modification to the bitcell array organization is necessary to implement BLADE, as all computation happens within the periphery. This greatly simplifies integration into existing SRAM fabrication design flows, where the most aggressive and complex design steps relate to the bitcell array. Finally, BLADE scales easily to larger cache arrays/subarrays without major loss of performance or energy savings, as LBL length is invariant with regard to subarray size, as will be discussed in Section 2.4.2.

2.3 Logic to Support Arithmetic Operations

While bitwise operations enabled by simple bitline logic may be sufficient to support applications such as cryptography or binary neural networks, more complex workloads necessitate arithmetic operations.

As discussed in Section 1.1.2, iSC architectures can perform either simple bitwise operations or arithmetic operations such as addition and multiplication. Supporting arithmetic operations introduces trade-offs in area overhead and latency as the BL logic becomes more complex. Carry logic may have to cross multiple BLs [27]. Alternatively, unorthodox methods of computation may be introduced, such as bit serial computation as in [41], which stores and operates on data in a transposed manner, limiting extra BL logic to a couple of latches, while also greatly increasing throughput. However, latency is also drastically increased, even quadratically for multiplication. Further, transposition presents challenges, requiring either extra hardware in the form of transposition functional units in the cache controller, 8T bitcells with extra BLs/WLs [42], which decreases subarray area efficiency as discussed in Section 2.2, or software transposition, introducing challenging programming complexity. For analog solutions, area overhead is a significant limitation, with reported overheads of 19% [37] and >30% [40], and the acknowledgement that WL Digital-to-Analog Converters (DACs) double array

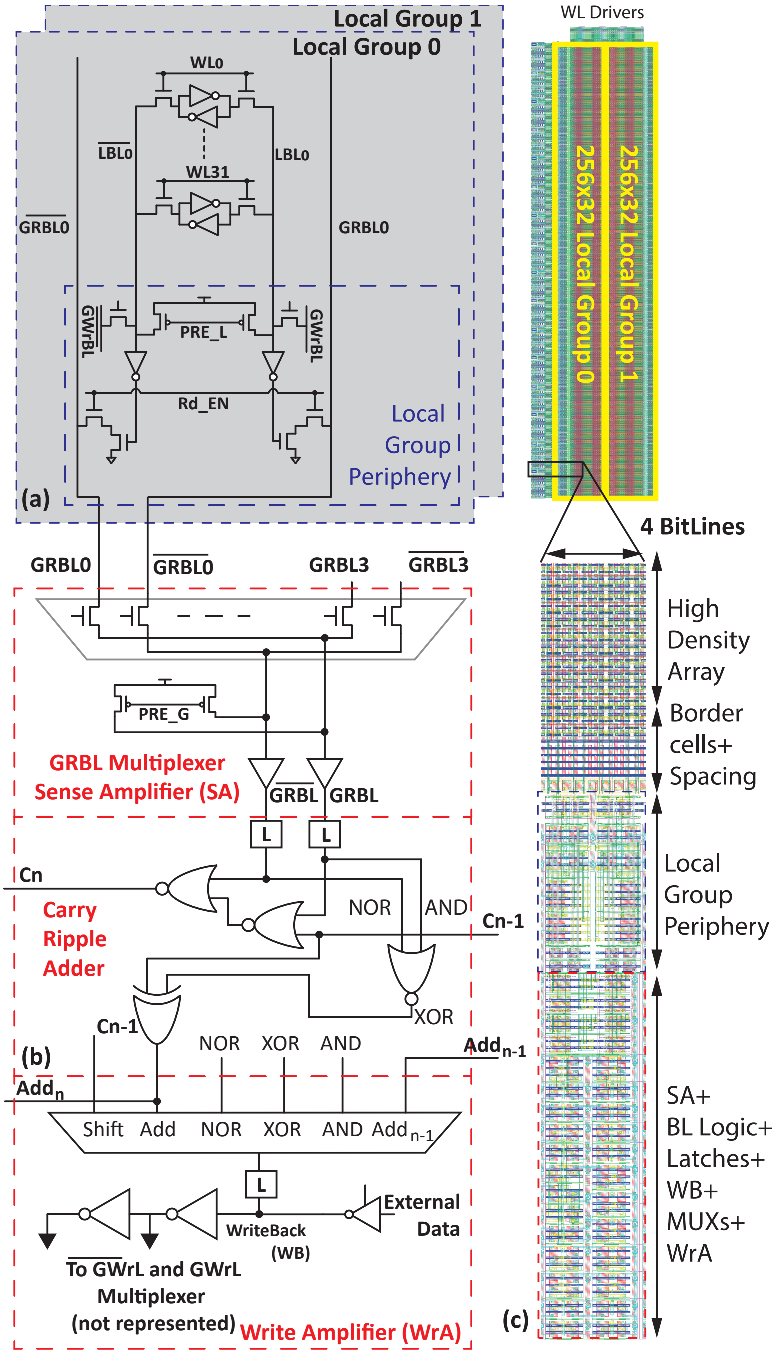


Figure 2.6: Layout of a 256×2×32 BLADE memory with schematic diagram of the BLADE periphery.

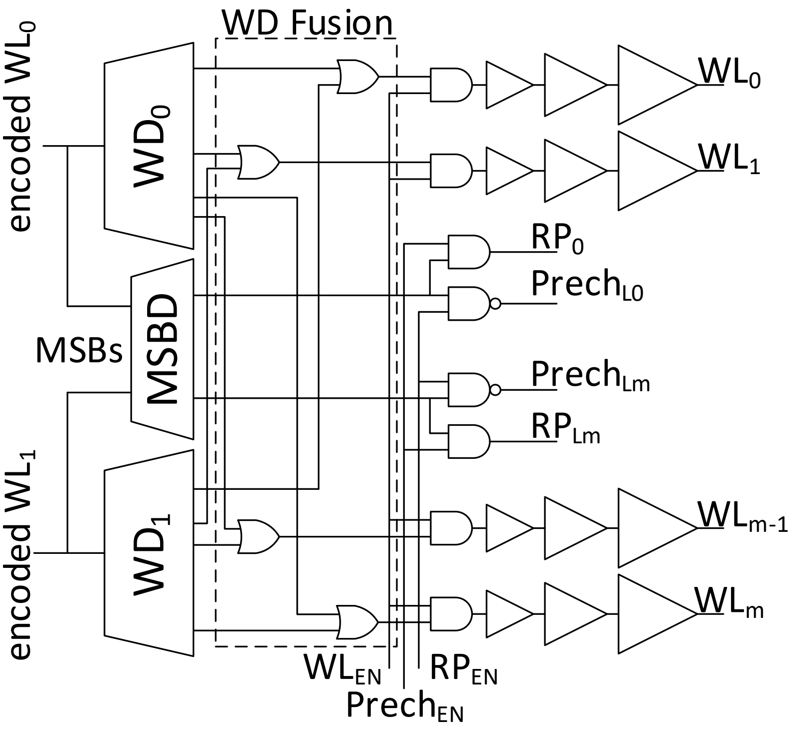


Figure 2.7: Block schematic of the modified WL decoder used in this work.

periphery [38]. Analog designs also perform inexact computations, which may be unacceptable for high precision applications, limiting generalizability, and are very susceptible to process variation, temperature, and aging [39].

In order to accelerate the widest range of workloads possible, BLADE supports common arithmetic operations, such as `addition`, `subtraction`, `multiplication`, `greater/less than`, and `shift`. These operations are implemented by augmenting the standard BL logic explained in Section 1.1.2 with carry and shift logic. By utilizing digital logic as opposed to analog, BLADE guarantees calculation exactness, necessary for high precision workloads.

2.3.1 Bitline Addition Architecture

Figure 2.6 illustrates BLADE's BL logic, as well as the transistor layout of a 4 way, 256x64 bitcell SRAM array with two local groups, each containing 32 WLs.

Algorithm 1 Modified add/shift multiplication for BLADE iSC computing.

```
Input: Op_0: Multiplier, Op_1: Multiplicand
Output: Res=Op_0 × Op_1
Latch Op_0
$i = \#Bits_{Op_0} - 1$
while $i \geq 0$ do
    $Res << 1$
    $tmp = Op_1 + Res$
    if $Op_0[i] = 1$ then
        $Res \leftarrow tmp$
    else
        $Res \leftarrow Res$
    $i--$
```

In order to support addition, a carry ripple adder is implemented underneath the array through the addition of two `nor` gates and a `xor` gate. Shift latches are also implemented within the BLADE controller to allow one cycle shifting. Each BL logic block receives a carry-in from the previous BL logic block, and provides a carry-out to the next block.

In order to drive multiple WLs as required by iSC operations, two WL decoders are utilized to simultaneously decode two WL addresses. A bitwise `or` is performed on the decoded addresses before driving the WLs, as illustrated in Figure 2.7. In an area-constrained environment, one decoder could be omitted in exchange for latches placed before the WL drivers, with the WL addresses decoded sequentially, similarly to the work presented in [27].

The implementation of addition logic enables many arithmetic operations, achievable through series of simple operations. For example, subtraction can be performed by negating the subtrahend by reading and storing the $\overline{\text{GBL}}$ value, then performing an addition between the operands with the first carry-in value set to 1.

Greater/less than is similarly computed by subtracting the operands and sensing the Most Significant Bit (MSB). Multiplication can be performed via a variation of the classic add and shift algorithm, as codified in Algorithm 1 and

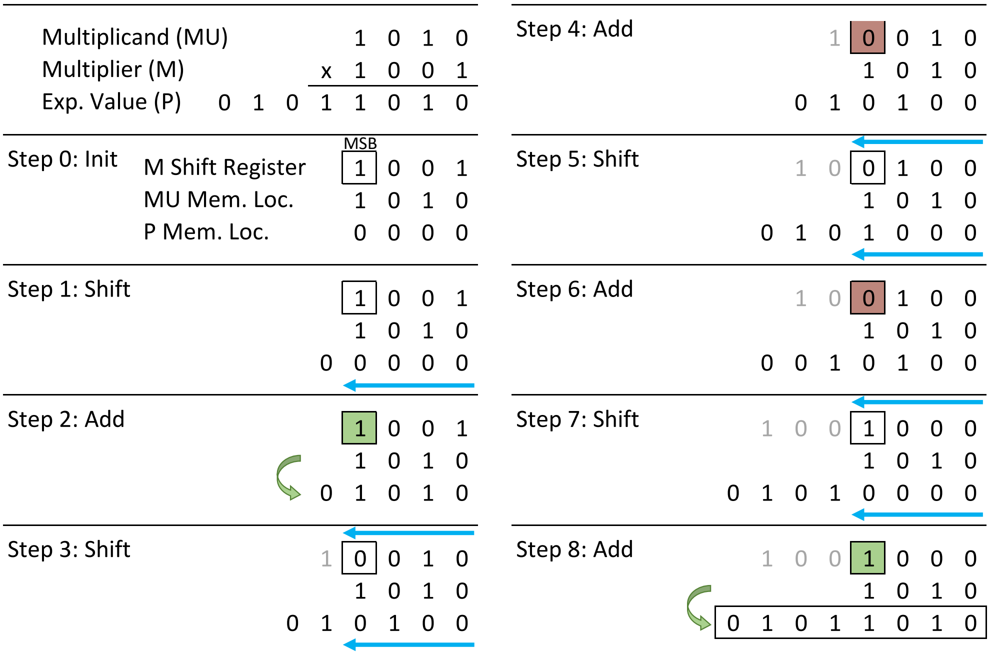


Figure 2.8: 4-bit shift-and-add multiplication as performed by BLADE.

illustrated in Figure 2.8, in which the product is shifted left each step (line 4) and the multiplier is processed from MSB to Least Significant Bit (LSB) to evaluate which partial products must be accumulated (lines 6-9).

As memory I/O (sense amplifiers and writeback logic) and BL logic are pitched under the memory array, cache associativity influences physical design layout. During layout analysis demonstrates that a mux-4 array (4 way associative cache, $2\mu m$ per 4 multiplexed BLs) enables the most efficient design for the I/O and BL logic, as illustrated in Figure 2.6-c. However, as the BL logic is pitched on $1\mu m$, it is also possible to pitch the BL logic in a mux-2 ratio, allowing twice the number of iSC computations per cycle, in exchange for approximately a 2.5x area overhead increase (nonlinear due to increased interconnect complexity). The system-level implications of such a configuration are explored in Section 2.7.

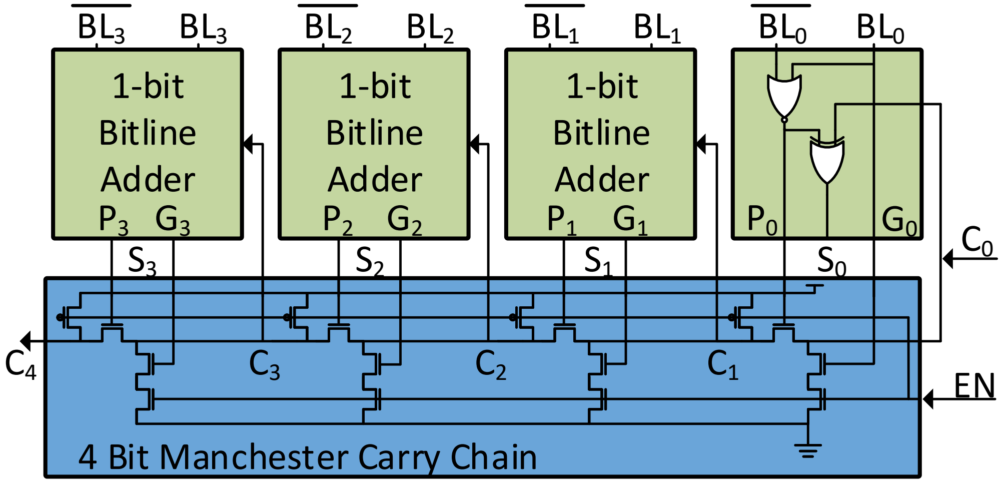


Figure 2.9: MCC architecture with reduced transistor count resulting from simplified Generate and Propagate signal generation.

2.3.2 Improving Operation Throughput

When performing arithmetic operations at longer word lengths, (e.g., 32/64 bit), the performance gains of BLADE are significantly mitigated by both the delay of the ripple carry adder, as well as (in the case of multiplication) the add and shift strategy, as demonstrated in Figure 2.10-a. To solve this problem, two optimizations are introduced to BLADE's design; (1) multiplication stage pipelining to reduce latency, and (2) a Manchester Carry Chain adder implementation to reduce the addition critical path.

2.3.2.1 Manchester Carry Chain (MCC)

At longer word lengths, the critical path of the ripple-carry adder substantially outpaces cache access times, reducing pipeline effectiveness. This effect can be mitigated by implementing a fast carry adder based on a dynamic Manchester Carry Chain (MCC) adder [64] in buffered 4-bits configuration, illustrated in Figure 2.9. The $\text{Generate}_{0:3}$ and $\text{Propagate}_{0:3}$ signals are generated with a single `nor` gate thanks to bitline computing, greatly reducing the area overhead typically associated with such an architecture.

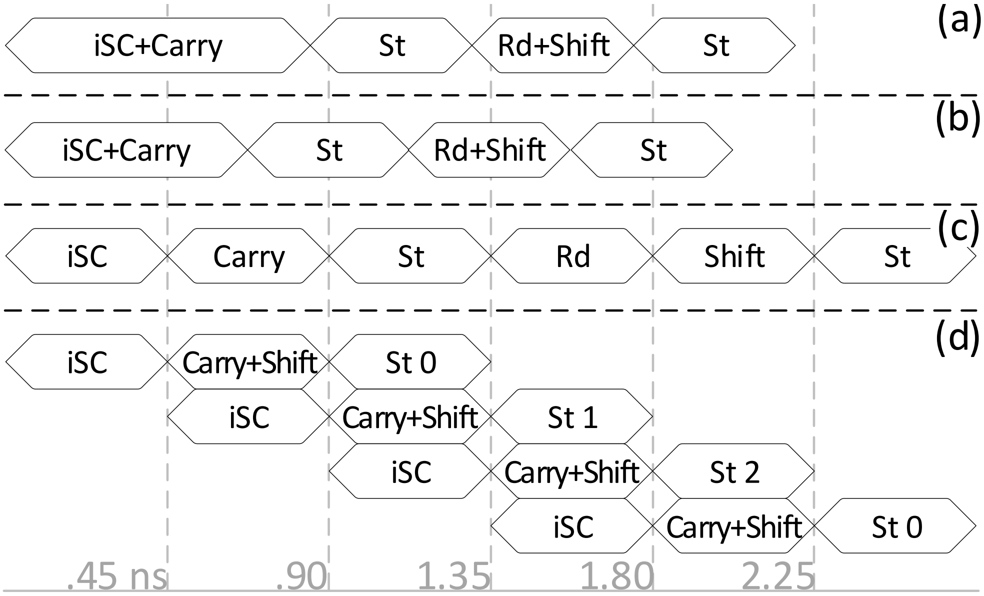


Figure 2.10: Timing diagrams of multiplication with (a) no optimization, (b) MCC, (c) pipelining, and (d) add-forward line.

Four MCC blocks are needed per 16 bitcell columns, as columns are mux-4 multiplexed, with the remaining space used to fit inter-MCC signal buffers as well as decoupling capacitors. Such a design provides nearly 80% performance improvement versus standard carry ripple adder at 1V, and a 54% improvement for 64-bit additions at 0.6V. Figure 2.10-b represents the reduced carry time on a multiplication shift-and-add cycle with an MCC adder.

2.3.2.2 Arithmetic Operation Pipelining

Shift and add multiplication can seriously mitigate performance if not properly implemented. In order to improve operation throughput, three optimizations that allow multiplication pipelining are proposed. First, latches after the sense amplifiers isolate the carry logic from the read and writeback stages, enabling these stages to be pipelined, as illustrated in Figure 2.10-c. Without latches, the iSC and ripple carry operation must be completed in a single step before performing writeback. Second, an add-forward line connecting the addition output of one BL logic block to the writeback stage of the next BL pair allows the shift and add operations to be performed in one cycle. Finally, as described in Section 2.2.2.1, BLADE-enhanced memory can perform a writeback and iSC operation simultaneously, if the writeback target line is in a different LG than those accessed for iSC. By first accumulating the product in three partial sums, then summing these partial values, full pipelining of the iSC, carry, and writeback stages of multiplication can be achieved, as illustrated in Figure 2.10-d. This strategy constrains the cache geometry to at least four LGs per subarray, one containing the multiplicand, which is accessed every cycle, and three containing the partial products.

2.4 Electrical Validation and Design Space Exploration

In order to verify the functionality of BLADE and the aforementioned optimizations, layout and simulation of BLADE is performed at the transistor level. A design space exploration is also carried out by varying cache geometry parameters in order to illustrate energy, delay, and area trends.

2.4.1 Functional Validation of BLADE

BLADE's layout methodology consists of implementation and simulation of the critical paths of the memory array (bitcells, WLs, global and local BLs), periphery (WL decoders and drivers), and iSC logic in 28nm bulk CMOS thin oxide transistors provided by TSMC's high performance technology PDK [65]. The periphery/iSC circuitry is implemented with the Low Voltage Threshold (LVT) technology flavor to optimize performance, while the memory array utilizes the Regular Voltage Threshold (RVT) technology flavor to limit static leakage. The designed bitcell is based on the work presented in [66, 67], achieving a bitcell pitch of 0.127μm, and the periphery is laid out with custom standard cells and pitched along the bottom and sides of the array with a spacing of 500nm between the array/periphery to account for any spacing required between SRAM and logic design rules. The simulated netlist contains >8000 elements, is simulated at 300K for 10,000 Monte-Carlo runs, and accounts for CMOS variability and post-layout parasitics. Memory and periphery signal propagation time is modeled by equivalent circuits for the lines with corresponding gates and extracted RC networks.

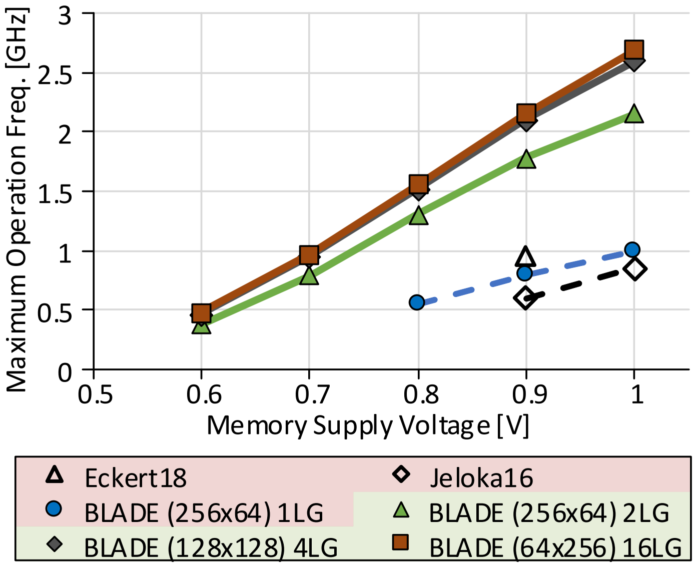


Figure 2.11: Maximum frequency of bitwise operations vs. memory supply voltage at 28nm CMOS.

Using this electrical characterization framework, BLADE area, timing, and energy values are extracted for different subarray geometries. Figure 2.11 shows the maximum clock frequency of BLADE vs. other proposed iSC architectures at different supply voltages. The dotted blue line represents bitwise iSC operation within a single LG, as described in Section 2.2.2.2, with similar performance to other iSC architectures. The utilization of LGs, on the other hand, enables significantly higher iSC operating frequencies, as illustrated by the green, gray, and red curves. As can be seen, the utilization of LGs improves operation frequency by 2.5x-3x depending on subarray geometry vs. standard iSC architectures and extends the operating range down to 416MHz/0.6V.

Finally, Table 2.1 details the worst-case energy/frequency values for different bitwise/addition iSC operations across different adder configurations. This table shows that, without optimization, 8/16-bit addition can be completed within 2.2GHz. However, reduced operating frequencies of 1.7/1.2GHz are necessary to complete 32/64-bit addition, respectively. By utilizing an MCC adder and carry logic pipelining, as explained in Section 2.3.2.2, a 2.2GHz operating frequency can be regained for 64-bit additions.

Table 2.1: Worst Case iSC Energy/Frequency Values in a 256×64 array with 2 LGs.

Operation		Rd	Wr	iSC	Add			
Width		Bitwise			8b	16b	32b	64b
E/op[fJ]		23.5	25.9	23.8	20.7	41.6	83.3	167
MMC Carry Prop. Time [ps]		-	-	-	64	130	258	512
Array Leak./op [fJ]		88.9					137	163
Freq.[Ghz]	(CRA w/o pipeline)	2.2	2.2	2.2	2.2	2.2	1.7	1.2
	(CRA w/ pipeline)	2.2	2.2	2.2	2.2	2.2	2.2	1.0
	(MCC w/ pipeline)	2.2	2.2	2.2	2.2	2.2	2.2	2.2

2.4.2 Subarray Design Space Exploration

There is a complex interrelationship between the parameters defining subarray geometry and the subarray's energy, delay, and area overhead. In order to demonstrate these relationships, the design space defined by a subarray's number of BLs, WLs, and LGs is explored, with results displayed in Figure 2.12-a. To more clearly illustrate tradeoffs between metrics, Figure 2.12-b illustrates the design space over a range of subarray geometries by normalizing the maximum value for each metric to one.

Area overhead is primarily influenced by the number of WLs belonging to each LG (LG size). In a 128x128 configuration, LG sizes of 16, 32, 64, and 128 result in area efficiencies of 55.6, 71, 84.4, and 91%, respectively. This is because larger LGs require less periphery per WL they contain, thus improving area efficiency.

Delay is influenced primarily by the LG size, and secondarily by the length of the WL. As LG size increases, parasitic capacitance increases and reduces switching time. Similarly, the parasitic capacitance of the WL increases delay, although this is partially offset by the reduced length of the GBL.

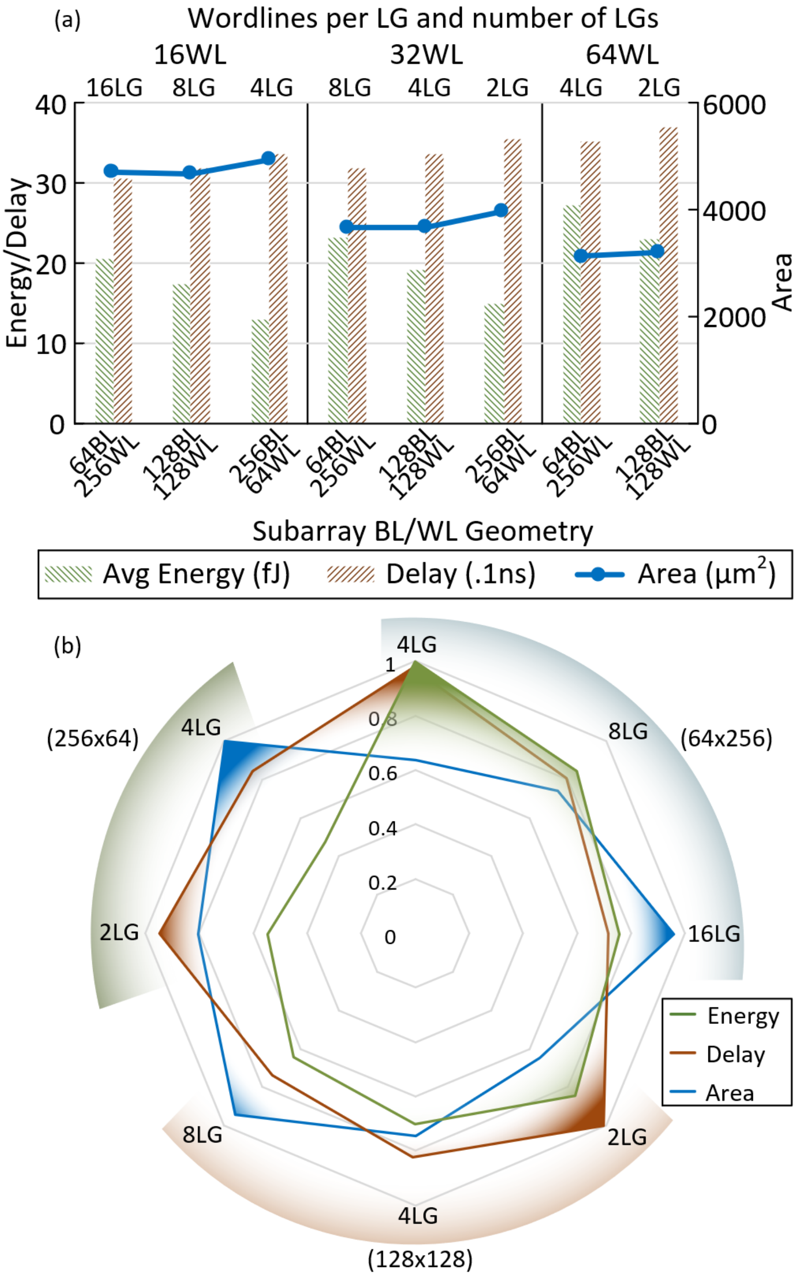


Figure 2.12: (a) Energy, area, and delay variations across cache geometries. (b) The normalized energy, area, and delay design space, with maximum values for each metric equal to 1.

Finally, energy is also influenced by LG size and WL length. Similarly to delay, energy consumption decreases with smaller LGs due to parasitic capacitance. In contrast however, energy decreases as WL length increases, as only one WL is activated for any number of BLs, meaning that the energy per bit decreases with increasing numbers of BLs.

System-level factors also play a role in deciding cache geometry. As discussed in Section 2.2.2.3, at least 2 LGs must be present in order to perform LBL-enhanced iSC operations. More generally, smaller LGs translates to less operand locality constraints on application data. Furthermore, 4 LGs are necessary to pipeline multiplication, as explained in Section 2.3.2.2.

As simulations indicate, different use cases may necessitate different cache geometries, whether it be low-power, low-area, or high-performance designs. When benchmarking BLADE in this chapter, a 128x128 subarray with 4 LGs is utilized. This subarray exceeds the 2.2GHz target operating frequency while maintaining a low energy consumption and achieving a 71% area efficiency, of which BLADE accounts for 8% area overhead. This design is comparable to industry designs, as area efficiency of ≥70% while maintaining reliability and good performance is optimistic for industrial high density SRAM arrays [68].

With BLADE's circuit level design implemented and characterized, we can now consider how BLADE can be integrated within the system architecture.

2.5 System-Level Integration and Functionality

The question of where to place an iSC architecture, how it integrates into the memory hierarchy, and how applications invoke it is a nontrivial problem. Problems such as virtual-physical memory translation, coherency, load/store consistency, interaction with standard memory functionality, and communication with the CPU must all be addressed. However, the majority of current literature focuses on the compute portion of the iSC architecture without discussing integration into the system architecture or system-level simulation. While some works discuss programmer/ISA support [24,25,32,41,69], only a few simulate their work in a full software stack environment [24,32].

BLADE is implemented within the cache hierarchy, as illustrated in Figure 2.13-a. Implementing BLADE within the cache reduces energy consumption resulting from data movement, as operand data will often already be loaded into the cache for use by the CPU. Also, in-cache implementation reduces area overhead by re-utilizing existing SRAM arrays.

As detailed in Section 2.3.1, in order to reduce area overhead, the BL logic is situated under a group of four multiplexed columns of bitcells. The simplest implementation of this architecture is to multiplex the columns' GBL pairs. However, this strategy comes with the downside of the inability for operations to be performed between the multiplexed columns. In the case of an in-cache implementation of BLADE as envisioned in this chapter, this corresponds to an inability to operate if operands are in different ways. This presents a problem, as control over in which way a cache block is stored is highly architecture dependent and transparent to the application programmer. This challenge can be overcome via way multiplexing within the LG periphery, as described in [70] and illustrated in Figure 2.13-b. Such an architecture allows operands stored in different ways to interact with each other, and also reduces area and energy overhead by 17.5%/22%, respectively. Finally, retaining way multiplexing functionality also enables parallel tag-data

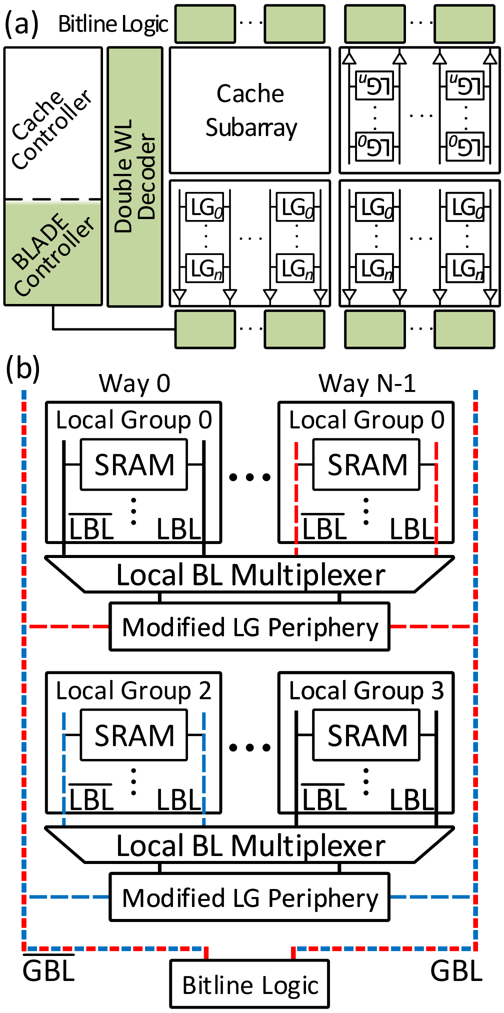


Figure 2.13: a) BLADE implementation within the cache, represented by highlighted boxes. b) BLADE implementation under Local Group (LG) way multiplexers.

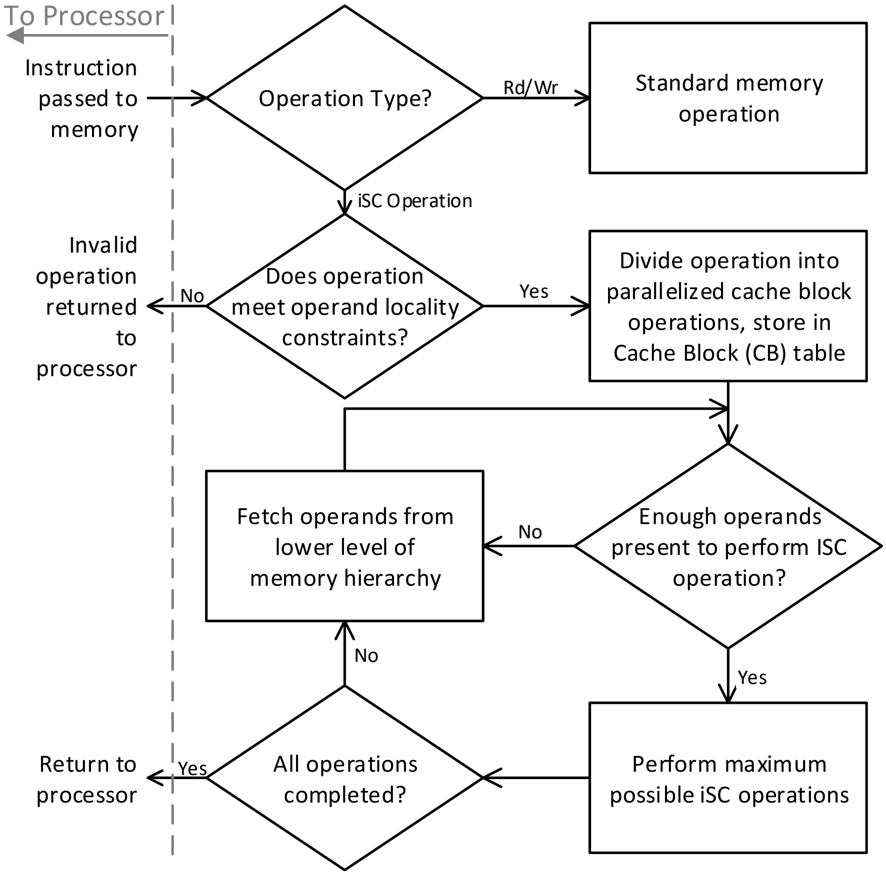

To Processor
Instruction passed to memory
Operation Type?
Rd/Wr
Standard memory operation
iSC Operation
Invalid operation returned to processor
No
Does operation meet operand locality constraints?
Yes
Divide operation into parallelized cache block operations, store in Cache Block (CB) table
Fetch operands from lower level of memory hierarchy
No
Enough operands present to perform ISC operation?
No
Yes
Return to processor
Yes
All operations completed?
Perform maximum possible iSC operations

Figure 2.14: Instruction flow in BLADE cache from time of issue by processor to operation completion.

access, a strategy in which the tag and data array are accessed simultaneously during a read, reducing overall read time at the cost of wasted energy in the event of a read miss.

Within the cache hierarchy, BLADE is specifically implemented within the private Level 1 (L1) cache, as this provides a favorable trade-off between area footprint and functionality and simplifies cache coherence considerations, in contrast to implementation in shared caches. BLADE can be implemented in the Level 2 (L2) and lower level caches, providing an increase in the number of possible parallel operations. However, such an implementation will increase coherence complexity in multi-CPU architectures, as modifications in shared lower level caches must be propagated to private caches. This can be accomplished by extending the coherence protocol to the BLADE controller, allowing it to invalidate cache blocks in the CPUs' private caches before performing computation, forcing the CPUs to reload the cache block.

Figure 2.14 illustrates the instruction flow from issue by the processor to the BLADE controller until operation completion and return to the processor. The following sections provide greater detail on iSC operation functionality.

2.5.1 Operand Locality Constraints

As mentioned previously, iSC operands must share BLs to be eligible for iSC operations. Sets that can interact with each other are considered local, and therefore the requirements placed upon operands can be called operand locality constraints. These constraints depend on the geometry of the cache, as factors such as cache size, subarray size, and associativity affect which sets share BLs. However, all variations in cache geometry can be abstracted to three constraints on the operand memory addresses:

- The offset bits between two operands must match, guaranteeing operand alignment within a cache block.
- A certain number of set LSBs must match, guaranteeing that the operands share the same subarray.
- A certain number of set MSBs must differ, guaranteeing that operands belong to different LGs, thus avoiding data corruption, as explained in Section 2.2.

The number of set LSBs that must match is quantified by the geometry value Val_{geo}, which specifies the cache geometry properties over which different sets do not share BLs, and therefore cannot interact with each other. This value is calculated according to the following equation:

$$Val_{geo} = \#_{banks} * \#_{subbanks} * \#_{subarrays} * \#_{spwl} \tag{2.1}$$

where each value represents a cache parameter. $N_{subarrays}$ equals the number of subarray rows in a subbank, and N_{spwl} is the number of sets per WL.

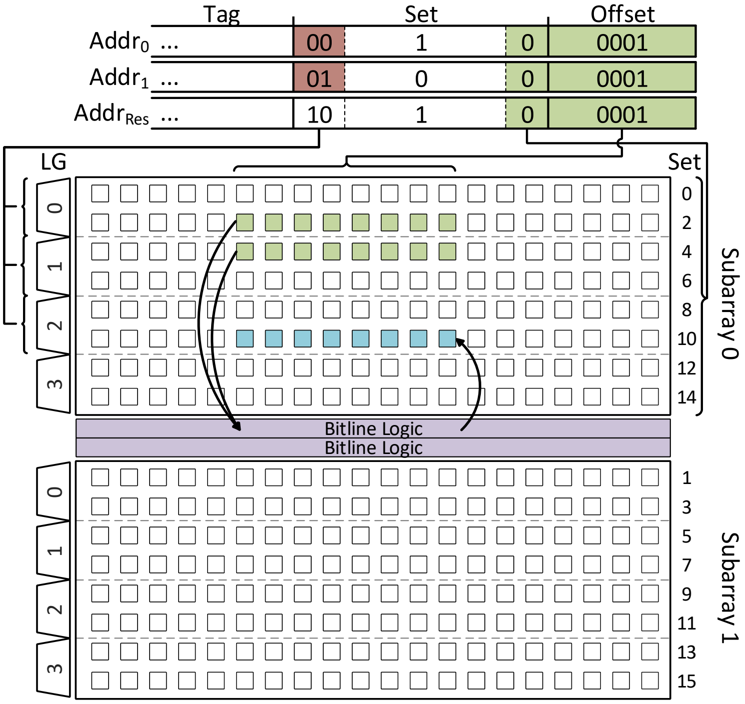


Figure 2.15: The operands in sets 0 and 1 have matching offsets and set LSBs, and differing set MSBs. Hence, they share BL logic, but not LGs.

By interleaving cache sets across these cache structures, as illustrated in the example array in Figure 2.15, the number of set LSBs that must match between operands is $log_2(Val_{geo})$. Moreover, any program compiled to function within certain Val_{geo} will also function on any cache geometry of the value of Val_{geo} or smaller. As the operands of this work's benchmark applications are page aligned, they can function on any cache geometry with a Val_{geo} value of up to 64.

Next, it is necessary that a certain number of set MSBs do not match. This constraint results from BLADE's utilization of LBLs to run at high frequency (2.2GHz@1V) while preventing data corruption, which necessitates that a certain number of WLs N_{LBL} in each subarray share one LBL pair, and two operands in an iSC operation must not share the same LBL pair. This geometry constraint can be guaranteed when N_{MSBs} of the set bits of each operand are different, with N_{MSBs} defined as:

$$N_{MSBs} = log_2\left(\frac{N_{sets}}{Val_{geo} * N_{LBL}}\right) \tag{2.2}$$

where N_{sets} is the number of sets in the cache and N_{LBL} is the number of WLs sharing an LBL pair. Note that N_{MSBs} cannot fall below 1, as this would result in all WLs in a subarray sharing one LBL pair.

Figure 2.15 illustrates a simple cache with N_{sets}=16, N_{banks} = $N_{subbanks}$ = 1, $N_{subarrays}$=N_{LBL}= 2, and N_{spwl}=1, resulting in Val_{geo} and N_{MSBs} values of 2. Therefore, one set LSB must match while one MSB cannot match.

The maximum number of simultaneous individual operations that can be performed per iSC operation can be calculated as follows:

$$\#_{i\_ops} = \frac{Val_{geo} * size_{cb}}{w_{i\_op}} \tag{2.3}$$

where w_{i_op} is the width of a single operation in bytes and $size_{cb}$ is the size of a cache block in bytes. Using Equation 2.3, one can calculate that the example cache in Figure 2.15 can perform 128 simultaneous operations, assuming a $size_{cb}$=64 bytes and w_{i_op}=1 byte.

In order to guarantee that operands meet data locality constraints, application data must be mapped to memory such that it is properly loaded to the cache at runtime. Guaranteeing proper operand alignment is a challenge that all iSC architectures face, with different solutions being proposed. For example, Jeloka *et al.* [24] handle this challenge by ensuring operands are page aligned and rely on future compiler extensions to guarantee this. On the other hand, Eckert *et al.* [41] propose the use of a special transpose unit in the cache controller for transposing and allocating data in-cache, and assume for their micro-benchmark that application data is laid out in DRAM such that it maps to the proper locations in SRAM. BLADE guarantees alignment by reserving a portion of memory at kernel boot time that can be mapped by an application at runtime. Because memory accessed in this way is guaranteed to map to a specific location in physical memory, one can ensure operand lo-

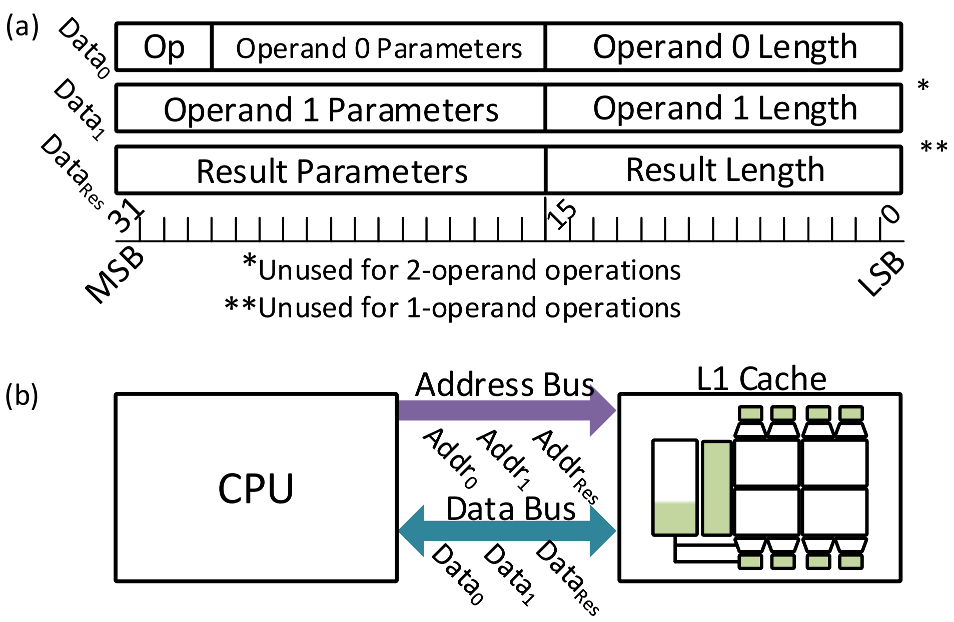


Figure 2.16: a) iSC operand parameter format. b) Passing BLADE operands in ISA implementation mode utilizes both the address and data bus.

cality by storing operand data in this reserved memory. The reserved memory is cacheable, contrary to standard reserved memory mapping.

2.5.2 iSC Instruction Passing and ISA Support

To enable utilization at the application level, BLADE must be made visible to the application developer. Two methods were explored for accomplishing this task; memory mapping BLADE, and integrating BLADE into the ARM Instruction Set Architecture (ISA).

2.5.2.1 Memory-Mapped Implementation

In the first implementation, the BLADE controller is memory addressable and can be memory-mapped and written to via pointer. This method has the advantage of being simple to implement, and its utilization will be familiar to embedded system application programmers. The number of instructions required to invoke BLADE depends on the number of operands required for the desired operation, with a pair of accesses representing each operand, one for the address and one for the operand parameters, as illustrated in Figure 2.16-a. The operand parameters consist of the following information:

- The opcode of the requested operation.
- The width of the operands (8/16/32 bit).
- Data unique to specific operations (e.g., the number of bits to shift for a shift operation.)
- The number of successive operations to be performed.

When all operands and addresses have been set, writing to the `start` register triggers the iSC operation. BLADE will delay the response packet until the operation completes, sleeping the application and freeing the CPU for use by other programs.

2.5.2.2 ISA Implementation

The second implementation involves adding a custom opcode to the ARM ISA. This is accomplished by identifying and allocating a reserved NOOP code in the ARMs's ISA as a special memory request that is routed to BLADE. This opcode can then be called in C/C++ using in-line assembly, which can be made transparent to the application developer behind an API. This method is more complex than the previously described method, but opens the path for compiler optimization, especially in regards to data placement to meet data locality constraints. It also reduces the number of memory access operations that need to be performed, as both the address and data bus are used to pass information to the BLADE controller, as illustrated in Figure 2.16-b.

Unlike the memory-mapped implementation, the CPU does not immediately sleep when issuing iSC commands in this way. It is therefore necessary to place a memory barrier before and after each set of commands to ensure correct memory ordering.

2.5.3 Issuing ISC Operations to Memory

Once received from the CPU, the BLADE controller breaks the iSC operation into operations up to a cache block in length, and stores them in an operation table. The operation table is responsible for confirming the operation's eligibility for in-cache computation as described in Section 2.5.1, fetching missing operands, and tracking the status of the operation. iSC operations are limited to a page in length to simplify address translation. Further, in cases where the number of specified operations is different between operands, address ranges of shorter operands will loop, allowing common operand reuse in cases such as multiplying a large input range by a small filter. Once all operands have been fetched and are present in the cache, the operation table issues the operation to the relevant cache subarrays. After all operations are complete, the BLADE controller alerts the CPU that the iSC operation has completed.

2.5.4 Fetching and Allocating Operands

The BLADE controller requests missing operands from memory. To simplify memory access, the BLADE control logic is implemented as a master module to the cache controller, similar to a CPU or Direct Memory Access (DMA) controller. iSC operations received from the CPU are forwarded to the BLADE controller. Then, all memory requests from BLADE are forwarded to the cache controller, which handles them as standard requests in regards to considerations such as coherency updating, evictions, or MSHR coalescing.

In order to guarantee correct memory coherency functionality, cache blocks subject to a snoop request will be evicted, and the cache controller will inform BLADE of the eviction. The block must be re-requested by BLADE in order to resume functionality. However, multi-cycle operations such as multiplication must be completed atomically, and therefore any snoop requests occurring during such an operation are rejected and reissued.

2.5.5 Integration in the gem5-X Architecture Simulator

As the complexity of computer systems have increased, proper simulation environments that accurately demonstrate the feasibility of an architectural innovation are paramount [71]. The complex interdependence of individual blocks within an architecture necessitate simulations that take into account a system-level view of the hardware and software environment. It is also beneficial to demonstrate the level of generalizability of an innovation, which requires a simulation framework that supports the ability to benchmark a wide range of applications easily. To this end, BLADE is integrated inside the gem5-X architectural simulator [54]. The gem5-X framework can be modified to model various combinations of CPUs and memory hierarchies alongside accelerators. gem5-X also provides a full system mode, which simulates a bare metal architecture on which a Linux software stack can be run, allowing any application to be run on top of an experimental architecture. With such a simulator, BLADE can be benchmarked on a wide range of applications, as will be demonstrated in Section 2.6.1.

In order to generate accurate performance statistics, timing values are extracted from BLADE's transistor level design and converted into CPU cycle values based on the simulated CPU's target clock frequency. These are injected into the gem5-X environment to perform realistic application-level simulations.

2.6 System-Level Benchmarking

In order to demonstrate BLADE's performance at the application level, BLADE is integrated into the gem5-X architectural simulator and profiled with a variety of emerging edge device workloads, enabling extraction of energy and runtime performance trends across a range of cache geometries [54].

2.6.1 Edge Device Workloads

As BLADE is targeted specifically for edge level devices, three applications that are becoming increasingly prevalent on edge devices are selected as benchmarks. These benchmarks demonstrate how BLADE is uniquely positioned for enhancing such devices.

2.6.1.1 Cryptography

One of the primary challenges the IoT industry faces as more and more sensitive data is stored and transmitted by edge devices is data privacy and security [72]. Indeed, lightweight cryptographic algorithms suitable for edge level devices are being developed [73], and many processors provide dedicated accelerators for cryptography [74]. Given these motivations, the Secure Hash Algorithm 3 (SHA3) integrity algorithm [75] is selected as a benchmark to illustrate that BLADE can provide low-power execution for such algorithms.

2.6.1.2 HEVC Video Processing

In 2018, Youtube and Snapchat combined accounted for over 20% of all mobile upstream traffic [5], and as more people broadcast their lives on social media, compression of upstream traffic will become a necessity on edge devices. The advent of 4K cameras on mobile devices will exacerbate this challenge, with more compression required to efficiently transmit. Kvazaar [76], one such application for High Efficiency Video Encoding (HEVC), is therefore benchmarked in order to demonstrate BLADE's capabilities in alleviating this problem.

2.6.1.3 Convolutional Neural Networks

The ubiquity of Convolutional Neural Networks (CNNs) as a solution for a variety of problems has led to increasing interest in implementing effective CNN solutions on edge devices, with both algorithmic [77, 78] and hardware [6, 79] innovations proposed. It is logical therefore to include CNNs amongst BLADE's application benchmarks. This benchmark is implemented with the Arm Compute Library (ACL) [59], an API from ARM designed to optimally utilize its NEON SIMD co-processor.

2.6.2 gem5-X Parameters

As stated in Section 2.5, iSC accelerators interact heavily with other components of the system architecture. In order to acquire accurate performance statistics while simulating such interactions, the gem5-X architectural simulator [54] with a full Linux software stack is utilized to benchmark BLADE. gem5-X is calibrated with the parameters outlined in Table 2.2, which emulates the ARMv8 A53 in-order core found on the ARM Juno development board [80], and runs an Ubuntu 18.04 LTS software environment that demonstrates less than 4% timing inaccuracy on profiling tests compared to physical

Table 2.2: Simulator Parameters

Processor	2GHz, 4 stage pipeline, ARMv8 ISA in-order core, 7 entry LSQ
Co-processor	NEON, 128-bit registers, 16 parallel 8-bit operations
L1-I Cache	32kB, 4-way, 1 cycle access
L1-D Cache	32kB, 4-way, 1 cycle access, BLADE
L2 Cache	mostly-exclusive, 1MB, 4-way, 6 cycle access
Memory	DDR3 2133MHz, 4GB
BLADE	Max 1024/128 bitwise/8bit simultaneous operations

hardware. BLADE timing values are integrated into gem5-X by converting delays calculated in Section 2.4.1 to cycle counts at 2GHz. The cache hierarchy utilizes a typical 64 byte WL length, with the cache geometry designed such that 1024 bitwise/128 8-bit iSC operations can be performed simultaneously. Performance comparisons are made against a NEON SIMD co-processor [28], a SIMD unit found on many edge devices.

2.6.3 McPAT Support

McPAT [81], an architectural framework for estimating the area and energy of a specified architecture, is utilized in order to estimate energy consumption of BLADE at a system level. For this work, McPAT is initialized with ARM Cortex-A53 architectural parameters [82]. This model is then augmented using the energy statistics specified in Section 2.4.1 to estimate the added energy consumption of BLADE operations. gem5-X provides traces of all CPU, memory, NEON, and BLADE operations, which are subsequently provided to McPAT to compute application runtime energy consumption for NEON and BLADE benchmarks.

2.7 Benchmark Results

During benchmarking, a wide range of hardware and algorithm parameters are explored to demonstrate trends in performance and energy consumption that vary depending on architecture design choices. Specifically, analysis is made of how 1) the number of iSC operations performed, 2) the cache's

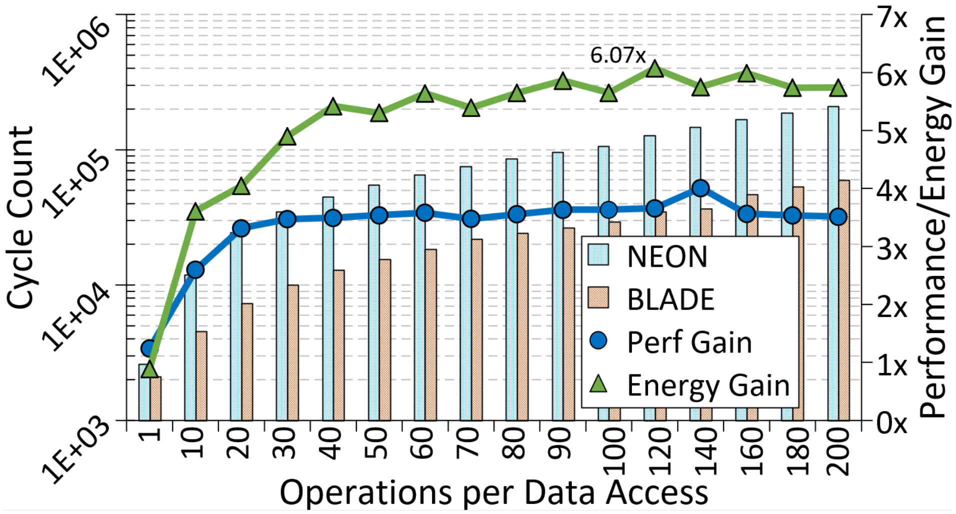


Figure 2.17: NEON/BLADE runtime/energy results on bitwise operations.

associativity, and finally 3) the cache size, affect performance and energy consumption. Also, analysis is performed on how optimization for arithmetic operations affects runtime of applications utilizing such operations.

2.7.1 Bitwise Operations/iSC Operation Count

In order to observe how the ratio between operations and memory accesses of an application affects BLADE performance, the block permutation kernel of the SHA3 algorithm is benchmarked. This kernel encrypts input data via a large number of bitwise operations, specifically `xor`, `shift`, and `and`, in a series of up to 24 rounds of permutation. By varying the count of bitwise operations being performed on input data, one can draw interesting conclusions about BLADE's effectiveness in accelerating such bitwise algorithms.

Figure 2.17 illustrates BLADE performance and energy improvement in comparison to NEON for bitwise operation/memory access ratios between 1 and 200 for 4096 bytes of data. As this figure shows, at lower ratios, memory access dominates function time. However, as operation count per data access increases, BLADE provides correspondingly increasing acceleration, saturating at a ~3.5x gain, achieved at 30 operations per access. As SHA3-256 performs over 400 operations per access, BLADE demonstrates strong applicability to

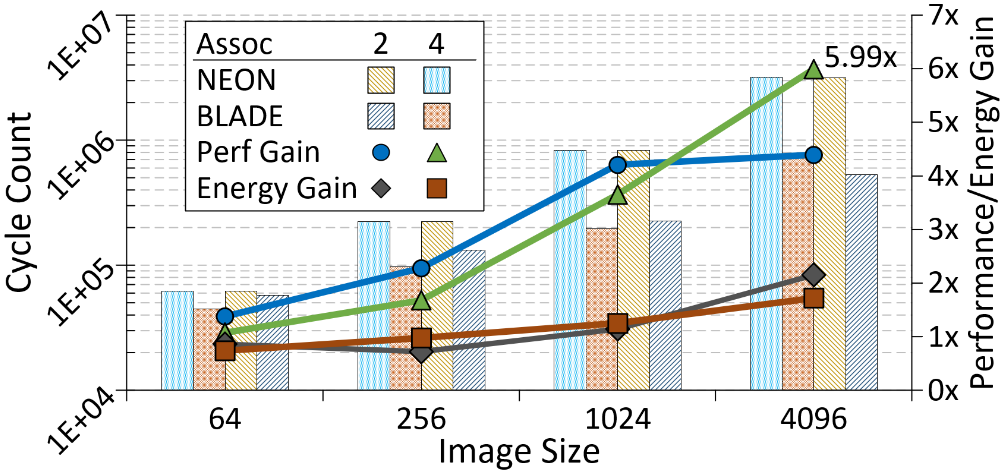


Figure 2.18: Runtime/energy results for NEON/BLADE on FIR operations.

such an application. BLADE's maximum demonstrated performance gain over NEON for bitwise operations is 4x.

Energy improvements follow a similar trend, with increasing energy gains that saturate at 40 operations per access. Energy improvement results from identical reasons as previously described; at higher operation/memory access ratios, energy consumption due to compute and L1-CPU data transfer become significant. BLADE eliminates L1-CPU transfers and reduces in-CPU operations, providing up to 6x energy improvement over NEON.

2.7.2 FIR Filter/Cache Associativity

In the next benchmark, the effects of cache associativity on performance are analyzed, as associativity directly impacts the number of simultaneous operations BLADE can perform, as discussed in Section 2.3.1. This is accomplished by varying the associativity between 2 and 4 ways at an equivalent cache capacity, while benchmarking Kvazaar's FIR filter function. This function uses four 8-tap FIR filters, filtering an input image first horizontally, then vertically, to produce 16 filtered outputs, which are then normalized and clipped between 0 and 255. Input images of four different sizes are accepted, with progressively increasing memory and compute requirements.

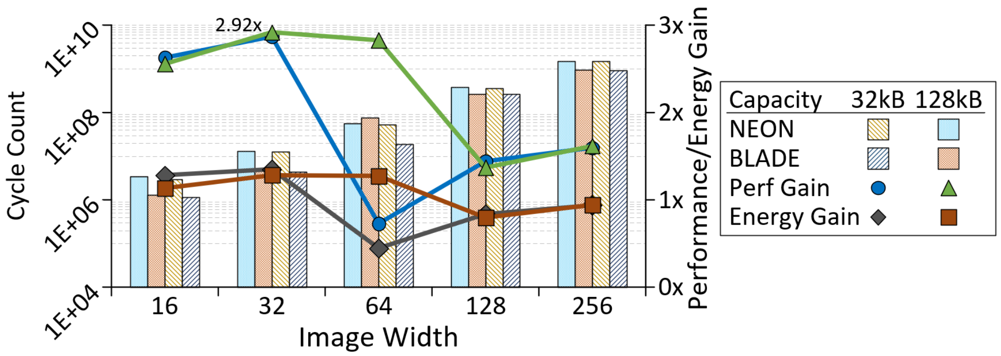


Figure 2.19: Runtime/energy results for NEON/BLADE on convolution operations.

Figure 2.18 illustrates performance and energy consumption differences resulting from the varying cache geometries. As this figure shows, a 4-way associative cache provides slightly higher acceleration over a 2-way associativity for tile sizes of up to 1024 pixels, as the wide 4-way cache results in less evictions of relevant data. However, at a tile size of 4096, compute requirements surpass those of memory, and therefore 2-way cache performance surpasses that of 4-way. This is because, assuming constant cache capacity, a 2-waache can perform twice the number of parallel operations. Overall, BLADE provides up to a maximum of 6x performance gain over NEON for FIR filtering. This can be attributed to the reduction of data movement between the memory hierarchy and processor, as well as the fact that the multiplication, normalization, and clipping all occur in-memory in a SIMD manner.

In contrast to the bitwise benchmark, FIR filter energy gain is more muted. This results from the fact that iSC multiplication is significantly more computationally complex than bitwise operations and therefore consumes more energy to execute. However, energy consumption due to L1-CPU data movement is still reduced, and BLADE still achieves a maximum of 2x energy reduction over NEON.

2.7.3 Convolution/Cache Capacity

In the next set of experiments, the effects of overall cache size on iSC performance is evaluated by benchmarking BLADE with a convolutional layer of a

CNN implemented in ACL. The convolutional layer has 32 input planes and 32 output planes, and performs 3x3 convolution at a stride of 1 with a padding of 1 to maintain equivalent dimensions. Input/output data is stored in 32-bit fixed-point notation and weights in 8-bit fixed-point notation, and input layer width is varied between 16 and 256 pixels. Two cache sizes, 32kB and 128kB, are utilized to demonstrate effects of cache size on BLADE effectiveness.

Figure 2.19 illustrates how cache size impacts performance. Performance results for each cache follow similar trends, where both implementations see performance drop-offs at larger image widths. This is due to the fact that at larger input widths, the resulting output plane does not fit entirely within the L1 cache, requiring a more complex kernel loop to satisfy operand locality constraints as described in Section 2.5.1, and resulting in increased data movement. However, by increasing cache size to 128kB, larger input widths can be accommodated while avoiding performance drop-off; in this case, 128kB caches can accommodate an input width of up to 64 pixels, as opposed to 32 for a 32kB cache. Furthermore, these results demonstrate the benefits of moving BLADE to higher capacity, lower level caches, as discussed in Section 2.5. Overall, BLADE demonstrates a maximum performance gain of 3x over NEON for a convolutional layer of a CNN.

Similarly to performance gain trends, a drop to below 1x energy gain is seen at pixel widths of 64/128 for cache sizes of 32kB/128kB, due to increased data movement resulting from ill-fitting kernels. Such trends indicate that operand locality is an important factor in ascertaining the effectiveness of a particular iSC architecture and cache geometry, Thus, not all kernels or applications are well suited to such an architecture. As discussed in Chapter 5, future work must be done to alleviate these constraints as much as possible. Ultimately, a maximum of 1.5x energy gain is achieved for the convolutional layer.

2.7.4 Arithmetic Logic Optimization

Lastly, analysis is made of how the arithmetic operation optimizations described in Section 2.3.2 affect FIR filter and convolution benchmark performance. Table 2.3 enumerates cycle counts for 8 and 32-bit iSC multiplications

Table 2.3: Cycles for 8/32-bit multiplication at different pipeline levels @2Ghz

Pipeline Level	Multiplication Cycle Count (8/32 bit)
No Pipeline	40/126
with Add-Forward	14/72
with Latches	24/66
Full Pipeline	15/39

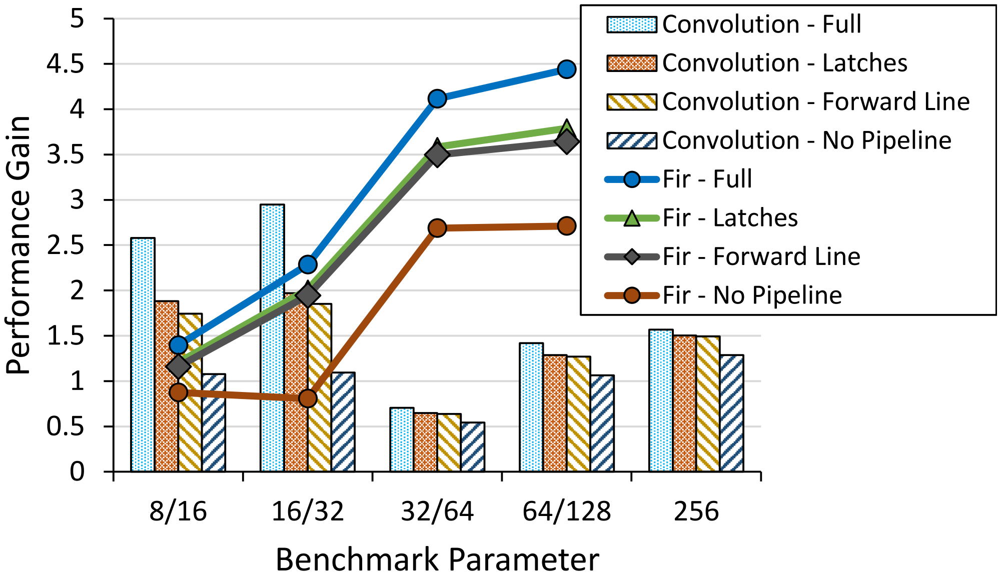


Figure 2.20: Performance trends for different levels of arithmetic pipelining.

for the benchmark architecture. These results show that 8-bit multiplication in fact completes in fewer cycles at 2GHz in an architecture without latches. However, the carry logic delay for 32 bits requires 2 cycles at 2GHz, resulting in a significant increase in cycle count for 32-bit multiplication. Figure 2.20 illustrates the effects of adding pipeline optimization to the BL logic for the FIR filter and convolution benchmarks, and demonstrates that pipelining provides significant performance improvement for a negligible area cost.

2.8 Summary and Concluding Remarks

iSC architectures show great promise in accelerating a variety of workloads, and are particularly interesting for edge devices due to their area and energy constraints. In this context, in this chapter I have presented BLADE. BLADE is an arithmetic iSC architecture whose utilization of industry standard 6T bitcell arrays enables easy integration into current SRAM fabrication flows, and its low power digital design makes it appropriate for accelerating emerging applications on edge devices. BLADE's functionality is validated from the electrical level up to the system level. At the electrical level, BLADE's enhanced cache design demonstrates how the use of local bitlines provides the best voltage/frequency ratio (0.6V/415MHz-1V/2.2GHz) of any 6T iSC architecture while maintaining a low area overhead of 8%. I presented a design space exploration of BLADE's electrical parameters, demonstrating tradeoffs between area, energy, and latency, dependent on cache parameters such as LG size and WL length. Next, at the architecture level, BLADE is integrated into the cache hierarchy of an in-order CPU, accounting for system-level interactions such as coherency and load/store consistency. Finally, BLADE is benchmarked on a full software stack with three emerging edge device workloads, and demonstrated 4x/6x, 6x/2x, and 3x/1.5x performance/energy gains over a NEON SIMD co-processor, thus validating our iSC design at the application level and demonstrating BLADE's effectiveness for edge level device acceleration. In the next chapters, I will explore further architectural modifications that can be applied to BLADE to further enhance its utility on edge devices.

3

Accelerating iSC Operations via Approximate Computing

BLADE, as described in Chapter 2, is capable of performing SIMD binary and arithmatic operations at native operating frequency and with low area overhead. One downside of BLADE is the shift-and-add nature of multiplications. While partially mitigated through operation pipelining as discussed in Section 2.3.2, multiplications still require at least N cycles to complete, where N is the width of the smallest operand. Reducing multiplication latency is critical to improving BLADE performance; therefore, this chapter will explore one method for accomplishing this using approximate computing.

The general objective of approximate multiplication is to reduce the area footprint, power consumption, and delay of a multiplier by approximating/simplifying aspects of the architecture. Kulkarni *et al.* [83], for example, modifies the Karnaugh Map of a 2x2 building block multiplier to halve its area. Another approximation method is the reduction of the partial product summation tree complexity, e.g., by performing partial product summation for only a portion of product MSBs, while combining the LSBs via a low-cost combinatorial method [84]. Thirdly, the full adder blocks can be approximated through a variety of methods [85]. Finally, the use of genetic algorithms to generate Pareto optimal approximate multipliers has been demonstrated to be effective [86, 87].

A challenge of approximate computing is the errors they introduce into applications. In this context, approximate computing is well suited to NNs, which are naturally robust against minor runtime perturbations [84]. As such, various approximate computing architectures have been proposed for enhancing NN efficiency. Even so, approximation error still degrades accuracy, necessitating various solutions such as limited neuron approximation [86, 88], or ensemble networks [89]. Accuracy degradation can also be mitigated through retraining while simulating Approximate Multiplier (AM) functionality; however, such retraining cannot be efficiently accelerated by employing GPUs as the properties of an approximate multiplier cannot be parallelized across many simultaneous multiplications, thus drastically increasing training time [87]. Further, many works do not utilize AMs uniformly across the network [86, 87], reducing generalizability in the case of hardware implementation.

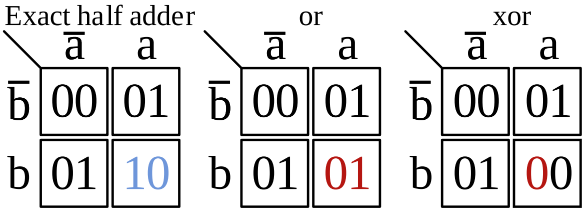
In relation to BLADE, we would like to find an approximate computing solution that can reduce the latency of multiplication while maintaining high application accuracy. To accomplish this, I introduce a CAPPIEM. CAPPIEM reduces multiplier latency by 2x in BLADE by enabling two multiplier bits to be assessed simultaneously, while incurring negligible area overhead. CAPPIEM also stands on its own as an approximate multiplier, demonstrating SotA improvements in latency, area, and energy consumption. By discarding the carry bit during partial product summation, 58% of the multiplier's full adders can be converted into `xor` or `or` gates, greatly reducing area, power consumption, and delay.

Beyond its area, energy, and latency savings, CAPPIEM also has a unique property in that it computes exact values if at least one input is Fibonacci encoded; that is, its binary form contains no consecutive ones. This property differentiates CAPPIEM from the vast majority of AMs, whose approximate output is dependent on both operands. This knowledge can be utilized to eliminate accuracy degradation due to errors injected by the AM by implementing a weight quantization strategy to guarantee exact multiplication during inference, which is coined Fibonacci Codeword Quantization (FCQ). First, FCQ encodes weights to Fibonacci code words, eliminating partial product summation errors. Then Incremental Network Quantization (INQ) is used to retrain the remaining weights to eliminate accuracy loss due to FCQ. As FCQ guarantees exact outputs from CAPPIEM, retraining can be perform without AM simulation, drastically reducing retraining time. Three INQ strategies are explored across three NNs, namely, ResNet-18, DenseNet-121, and Squeezenet 1.0, on the CIFAR-100 dataset. Benchmark results demonstrate the possibility to achieve full FCQ with only 0.4%, 1.1%, and 1.7% accuracy degradation, respectively.

The contributions of this chapter are as follows:

- CAPPIEM is an Approximate Multiplier (AM) that halves BLADE multiplication latency while incurring negligible area overhead.
- CAPPIEM also functions as a standalone AM that replaces 58% of an 8-bit standard multiplier's full adder operations with `or` logic. CAPPIEM is implemented in 65nm TSMC CMOS and provides area/power-delay-product reductions of 73/43%, respectively, while maintaining a low mean relative error distance of 0.054 in comparison to other AMs.
- Fibonacci Codeword Quantization (FCQ) is a strategy for weight quantization such that weights produce exact results when multiplied via CAPPIEM. FCQ reduces retraining time by 300x compared to retraining with other AMs.
- Incremental Network Quantization (INQ) is used to recover accuracy lost due to FCQ. Multiple INQ strategies are explored across three benchmarks, namely, Squeezenet 1.0, DenseNet-121, and ResNet-18, demonstrating full FCQ with accuracy losses of 0.4%/1.1%/1.7%, respectively.

The remainder of this chapter is organized as follows. Section 3.1 details the CAPPIEM implementation. Section 3.2 explains how FCQ eliminates AM errors. Section 3.3 details the proposed INQ strategy for regaining accuracy. Sections 3.4 and 3.5 detail the benchmarking process and analysis, and contextualize this work among other approximate NN publications. Finally, Section 3.6 concludes this chapter.



Exact half adder

	$\bar{a}$	a
$\bar{b}$	00	01
b	01	10

or

	$\bar{a}$	a
$\bar{b}$	00	01
b	01	01

xor

	$\bar{a}$	a
$\bar{b}$	00	01
b	01	00

Figure 3.1: Karnaugh maps for an exact half adder, as well as `or` and `xor` gates. While `xor` gates compute the sum value exactly in all instances, `or` gates provide the least error when the carry value is considered.

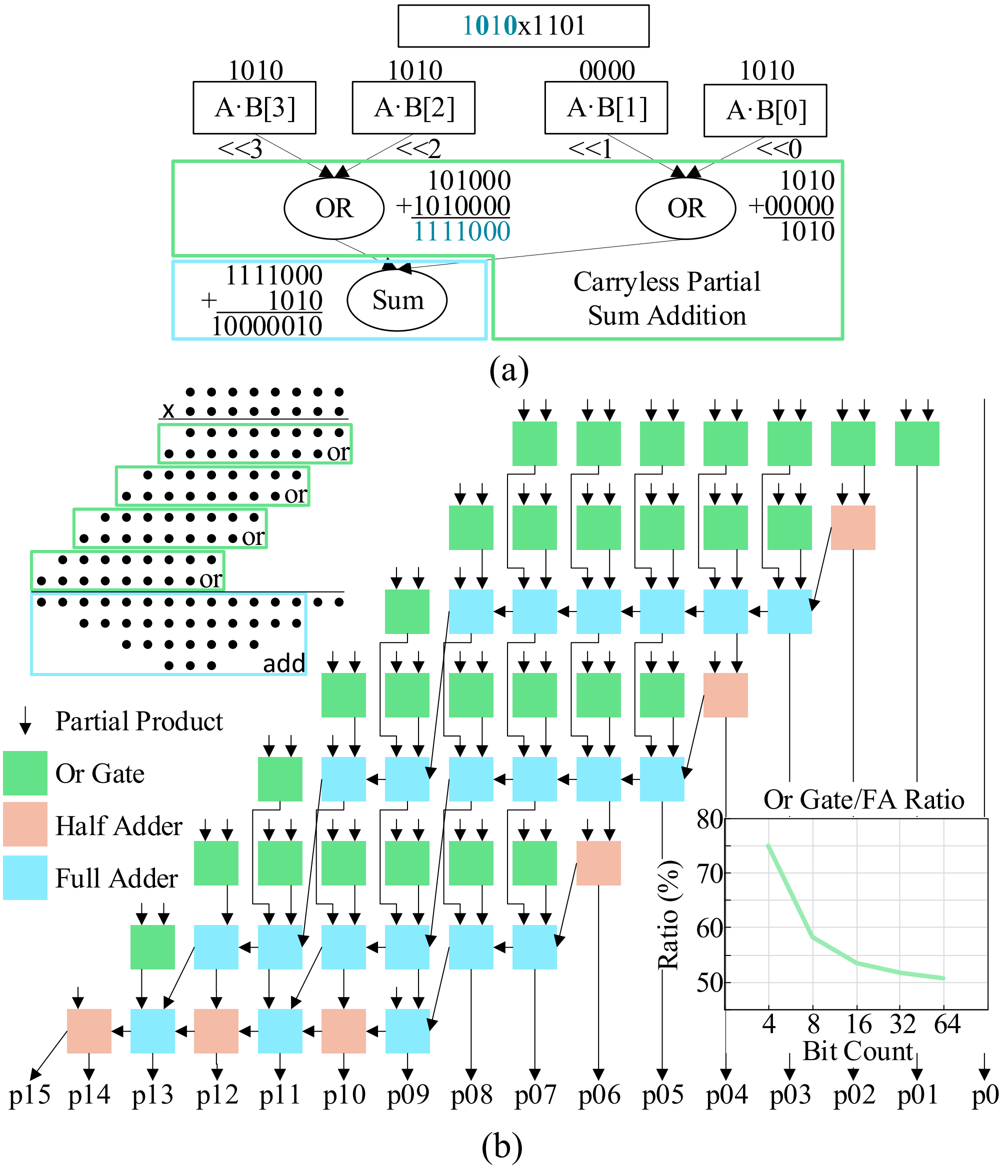
3.1 CAPPIEM: A Carryless Partial Product Inexact Multiplier

CAPPIEM is best illustrated by first describing it in respect to a traditional carry-lookahead adder tree, then explaining how it is implemented in BLADE.

CAPPIEM is implemented in a standalone-adder by simplifying a portion of Full Adders (FAs) within the adder tree to `or` or `xor` logic, as shown in Figure 3.2-a. This modification greatly reduces multiplier area, power consumption, and delay, at the cost of potentially introducing approximation error. A weight quantization methodology for avoiding such errors is described in Section 3.2.

3.1.1 Selection of Reduction Operator

When simplifying the FAs, two viable replacement options consist of `or` and `xor` gates. A `xor` gate advantageously produces only one incorrect bit with inputs $ab = 11$, as seen in Figure 3.1. In contrast, `or` gates require fewer transistors and produce a closer exact value for $ab = 11$. Indeed, one finds that for all input combinations for an 8-bit multiplier, the Mean Relative Error Distance (MRED), or average distance between every approximate product and its expected value, is 0.99 for a `xor` based AM, while the MRED of an `or` based AM is only 0.054. An `or` vs. `xor` gate implementation of CAPPIEM is also simpler to implement in BLADE, requiring only two extra transistors. This work therefore utilizes `or` gates for partial product reduction [90].



(a)

(b)

Figure 3.2: (a) 4-bit multiplication with carryless partial product summation enabled by `or` gates. (b) 8x8 adder array with FA gates replaced by `or` gates to reduce area and power consumption. The ratio of replaced adders approaches a limit of 50% as bit width increases.

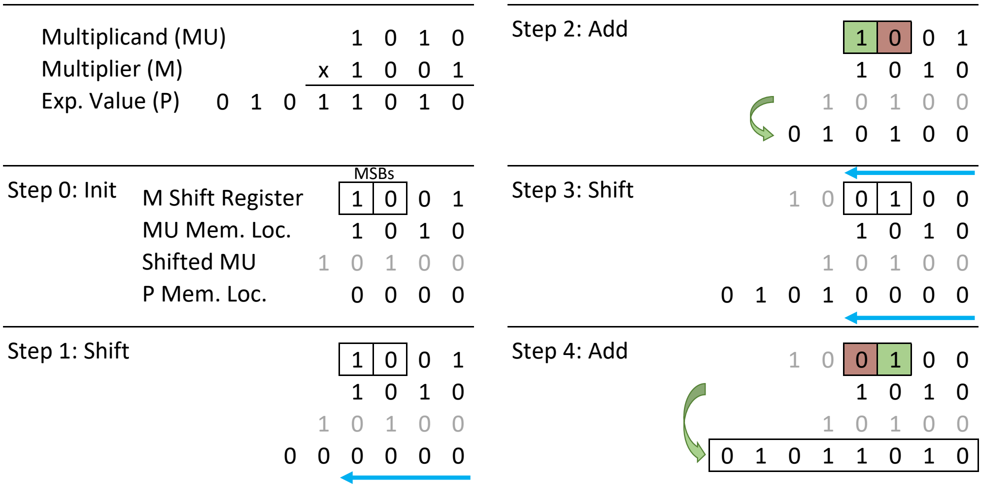


Figure 3.3: 4-bit shift-and-add multiplication as performed by BLADE. If the highest MSB is 1, a left-1 shifted multiplicand will be added to the partial product. If the second highest MSB is 1, the original multiplicand will be added. If both are 1, a `or`-ed combination of the two will be added.

3.1.2 CAPPIEM Hardware Implementation

Figure 3.2-b illustrates the CAPPIEM architecture applied to a carry save multiplier. Initially, an n-bit carry save architecture contains $(n-2)*n$ FAs and $n-1$ half adders. To implement CAPPIEM, the adders responsible for partial product summation are replaced with `or` gates, visible as green boxes. The remainder of adders, symbolized by blue and orange boxes, accumulate the intermediate values into the final product and are left unchanged. Hence, the number of FAs replaced can be calculated via the equation $\frac{n^2-n}{2}$. At 4 bits, the ratio of replaced FAs is 75%; this ratio decreases to 50% as multiplier width is increased.

3.1.3 Implementation of CAPPIEM in BLADE

As BLADE utilizes a shift-and-add strategy to perform multiplication, an iterative algorithm must be devised to utilize the concept of CAPPIEM. Figure 3.3 illustrates a sequential carryless partial product kernel that can be easily implemented with minimal change to BLADE. The main difference to the original shift-and-add kernel presented in Algorithm 1 is that, instead of only

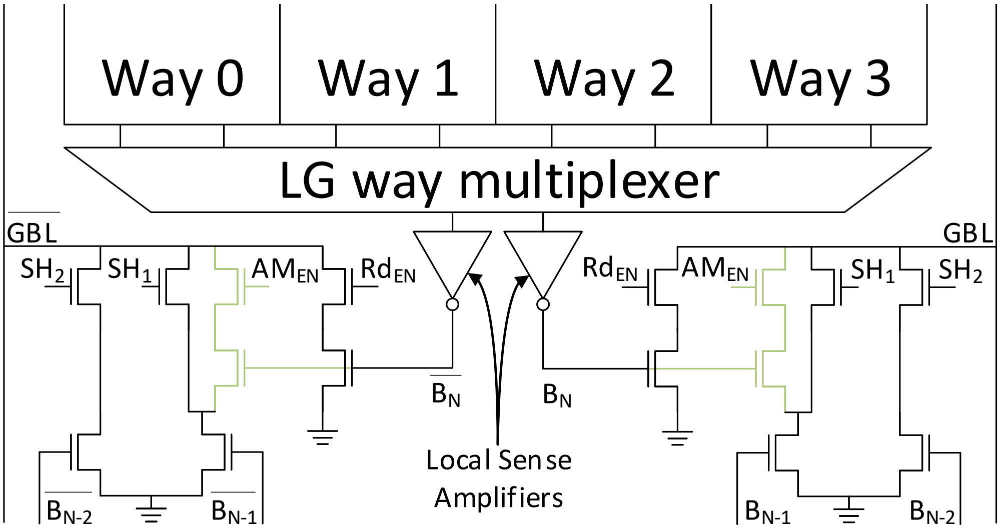


Figure 3.4: Embedded shift logic with CAPPIEM extension in green. Extension performs an `or` operation between the current and previous bit of the operand.

considering the first MSB of the multiplier, the first two MSBs are considered. If the first MSB is one, the multiplicand is left-shifted one and added to the partial product. On the other hand, if the second MSB is one, the unshifted value of the multiplicand is added. If both MSBs are one, the multiplicand is `or`-ed with a one-bit left-shifted copy of itself before being added to the partial product. The approximation occurs when both MSBs are one. Note that Figure 3.3 separates the shift and add operations into two steps to easier highlight the addition functionality of CAPPIEM; these steps are collapsed into one using the shifting logic described below.

CAPPIEM is implemented in BLADE by extending the embedded shift architecture demonstrated in [70], which in turn is an extension of the feed-forward line BLADE optimization presented in Section 2.3.2.2, and is illustrated in Figure 3.4. The base architecture, similar to that found in [70], enables left shifting of up to two places by connecting the output of the LG to the neighboring local sense amplifiers, resulting in a shifted output on the GBLs. By adding two transistors to this base architecture, it is possible to perform an `or` operation between B_N and B_{N-1}. This enables two bit shifted

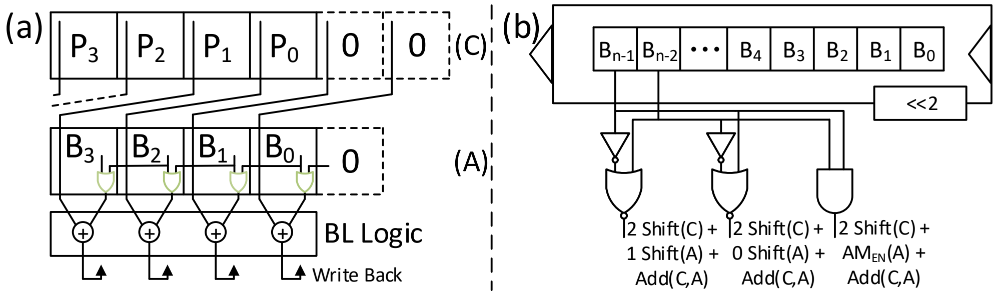

(a)
P3
P2
P1
P0
0
0
(C)
B3
B2
B1
B0
0
(A)
BL Logic
Write Back
(b)
Bn-1
Bn-2
B4
B3
B2
B1
B0
<<2
2 Shift(C) + 1 Shift(A) + Add(C,A)
2 Shift(C) + 0 Shift(A) + Add(C,A)
2 Shift(C) + AMEN(A) + Add(C,A)

Figure 3.5: Embedded shift logic with CAPPIEM extension in green. Extension performs an `or` operation between the current and previous bit of the operand.

addition, as illustrated in Figure 3.5-a and resulting in the shift-and-add multiplicative kernel described in Figure 3.3. This algorithm produces an output identical to the standalone AM described above.

Figure 3.5-b illustrates the shift control logic that controls the in-memory shifting for the partial product and the multiplicand. In each shift-and-add cycle, the partial product will shift two places, while the multiplicand will shift one or zero spaces if either the first or second MSB of the multiplier is 1, respectively. If both bits are 1, the `or` logic embedded in the LG periphery `or`-s the multiplicand with itself left-shifted one.

Interestingly, eliminating the extra transistors in Figure 3.4 and utilizing the shift control logic described in [70] results in a similar AM that utilizes `and` gates instead of `or` gates to perform carryless partial summation. Such an AM makes little sense; however, the quantization strategy described in the next sections works equally well for such an AM; it may therefore be beneficial to consider such an AM if the use case is well defined and area overhead is a tight requirement.

One setback of this (and indeed, most) AMs is their injection of approximation errors into applications. It would be preferable to be able to predict for which inputs an AM will produce exact or erroneous results. For CAPPIEM, such prediction is possible, a fact which is exploited in the next section to enhance neural networks with approximate multiplication while maintaining a high output accuracy.

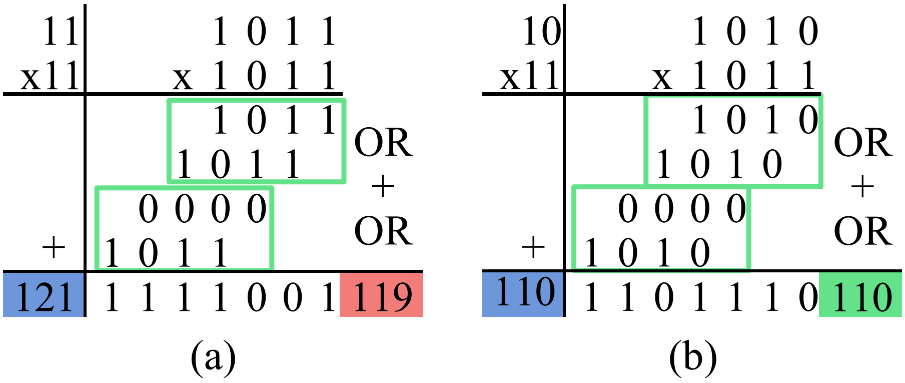


Figure 3.6: Naive carryless partial product multiplication will lead to errors and reduced accuracy (a). Such errors can be avoided by quantizing NN weights according to Fibonacci code words (b).

3.2 Fibonacci Code Word Quantization for NNs

3.2.1 Motivation

The AM proposed in Section 3.1 provides significant reductions in area, power consumption, and delay, which will be quantified in Section 3.5; however, its naive utilization in a neural network would introduce unacceptable accuracy degradation. It has been demonstrated previously that retraining an AM-enhanced NN recoups lost accuracy [91]. However, in order to retrain the network, it is necessary to simulate the functionality of the AM, for example via a lookup table [87]. AM simulation while retraining prohibitively increases training time (15 days for ResNet-18 retraining [87]) by precluding the use of hardware components, such as vector processors or GPUs. While this has proven a challenge for previous works, CAPPIEM enables a novel method of retraining without AM simulation via Fibonacci code word quantization.

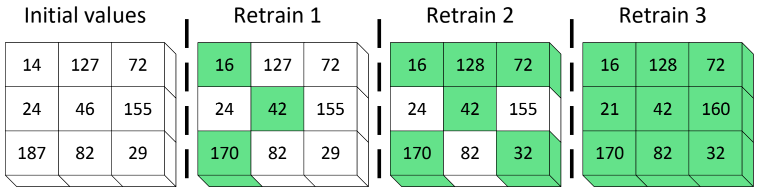
3.2.2 Countering Approximation Errors Via Fibonacci Code Word Quantization

Approximate multipliers by their nature introduce errors into the product of the operation. For CAPPIEM, the introduced error is illustrated in Figure 3.6-a, namely, if two partial products contain overlapping ones, the resulting value will be incorrect. To avoid such errors, it is proposed to quantize weights such that errors will not occur when reducing partial products. This is accomplished by quantizing weight values to the closest Fibonacci code word [92], that is, the closest value such that no consecutive ones appear in the binary representation of at least one of the operands, as illustrated in Figure 3.2. The result of such a quantization is that the sum of any two partial products is equivalent to a bitwise `or` operation between them, enabling them to be summed exactly. Such a quantization and multiplication is illustrated in Figure 3.6-b. Importantly, *only one* multiplier input needs be a Fibonacci code word to guarantee an exact output; the other input can be any value between 0 and $2^n - 1$. This qualification is necessary for utilization in a NN without needing to also modify layer inputs. This method of quantization is coined Fibonacci Codeword Quantization (FCQ).

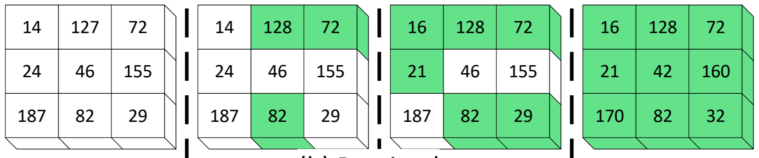
3.2.3 Quantization Parameters for Fibonacci Code Quantization

FCQ is based on the low-precision general matrix multiplication (gemmlowp) [93] method. In gemmlowp, a scale and zero-point value for the weights of each layer are calculated such that the weight matrix can be scaled between minimum and maximum fixed-point values and the real value of 0 is exactly representable. FCQ builds on gemmlowp by further quantizing the fixed-point values to Fibonacci code words. A few considerations must be made to ensure that FCQ can be co-implemented with gemmlowp.

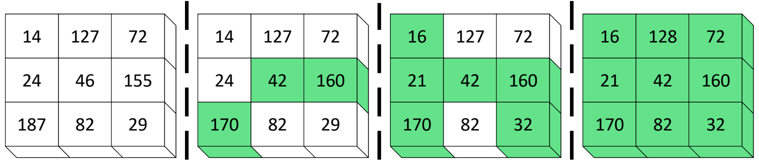
First, when quantizing a network, the range of values to which the weights are quantized must be considered. In gemmlowp, this is typically the range of signed values a given n-bit binary value can represent, e.g., -128 to 127 for 8-bit signed values. In the case of FCQ, asymmetric quantization is used, with a minimum value of 0 and a maximum value $quant_{max}$ of $2^n - 1$. This is necessary as small two's complement negative values contain many consecutive ones and thus are poorly represented as Fibonacci code words. Bias values are not encoded to Fibonacci code words as they are not involved in multiplication.



(a) Random



(b) Proximal



(c) Distant

Figure 3.7: Three incremental quantization strategies for Fibonacci incremental retraining; (a) Random, (b) Proximal, and (c) Distant. In each retraining step, a fraction of weight values are quantized to the nearest Fibonacci code word and frozen, represented by green boxes.

FCQ imposes further constraints on the upper range of quantizable values, as the maximum possible unsigned n-bit value F_{max} a weight can take is '10' repeating for $n/2$ bits (e.g., 10101010 or 170 for 8-bit values), with any value above this being clamped to F_{max}. In order to maintain weight value variance while preventing an excessive quantity of weights from being clamped to this maximum value, a q_{max} value of $\frac{(quant_{max}+F_{max})}{2}$ is selected, midway between the max quantized and max Fibonacci values (212 for 8-bit values).

Finally, in conjunction with asymmetric quantization, using max pooling layers in contrast to average pooling layers provides higher network accuracy, as such layers are less impacted by the elimination of negative values.

Extreme forms of quantization, e.g., FCQ as presented here, typically induce unacceptable levels of accuracy loss if implemented in isolation. However, this barrier can be overcome through the use of Incremental Network Quantization (INQ) [94] to recover lost accuracy due to FCQ.

Algorithm 2 Incremental Fibonacci code quantization and retraining flow.

Input: f_model: Float Model, strategy: INQ Strategy, q_steps: INQ Steps, r_epochs: # Retraining Epochs

Output: Fibonacci Code Quantized Model

```
1: def fib_quantize(f_model, strategy, q_steps, r_epochs):
2:     q_model = quantize(f_model)
3:     for i = 0; i < len(q_steps); i++ do
4:         q_model = fib_enc_and_freeze(q_model, q_steps[i])
5:         f_model = dequantize(q_model)
6:         for j = 0; j < r_epochs; j++ do
7:             f_model = train(f_model)
8:         q_model = quantize(f_model)
9:     return q_model
```

3.3 Improving Accuracy through Incremental Network Quantization

In INQ, weight values are incrementally quantized to a range or set of values and then frozen while the remainder of the network is retrained to mitigate the subsequent loss in accuracy. In previous works, INQ as a method to recover lost accuracy due to approximate multiplication would lead to prohibitively long training times, as the AM must be simulated via a lookup table during training [87]. However, as discussed in Section 3.2.2, the AM presented in Section 3.1 necessitates that only one input value be a Fibonacci code word, while the other input may take any value. This characteristic enables us to retrain the network without needing to simulate the AM, as Fibonacci quantized weights are guaranteed to produce exact products.

Algorithm 2 illustrates the proposed methodology for performing FCQ via INQ. At each iteration (line 3), the algorithm quantizes the network to 8 bits, performs FCQ on a fraction of the weights, freezes their values (4), then converts the model back to floating point (5). The remaining values are then retrained to regain accuracy (7). These steps are repeated until FCQ has been applied to all weights.

Table 3.1: Cumulative Fraction of Weights to Quantize at Each Incremental Quantize-and-Retrain Step

Strategy	**Cumulative Fraction of Weights Quantized**
Random	[0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5, 0.55,0.60,0.65,0.7,0.75,0.8,0.85,0.9,0.95,1.0]
Proximal	[0.3,0.4,0.5,0.6,0.7,0.8,0.85,0.9,0.95,0.98,0.99, 0.995,0.998,0.999,0.9995,0.9998,0.9999,1.0]
Distant	[0.001,0.0025,0.005,0.01,0.025,0.05,0.1,0.15, 0.2,0.25,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

3.3.1 Incremental Quantization Strategies

In order to implement INQ, a strategy to iteratively quantize weights must be selected. three such strategies are explored in this chapter, namely, random, proximal, and distant, as illustrated in Figure 3.7.
Random allocates weights randomly to quantization steps. This has the advantage of being easy to implement, but is simplistic and can be optimized.
Proximal allocation first quantizes weights closest to the nearest Fibonacci code word. This strategy results in less perturbation in the early training stages, reducing the chance of placing the network in an unrecoverable state.
Distant allocation is the inverse of the proximal strategy. This strategy performs most retraining in the early steps, allowing more potential for recovering accuracy, as fewer weights are frozen during steps of greatest quantization.

For each quantization strategy, the cumulative fraction of weights to be quantized at each step is defined. For the random strategy, the fraction of weights increases uniformly from 0.0 to 1.0. For proximal, an aggressive approach is taken towards initial quantization, as the first weights quantized are closer to a Fibonacci code word. Conversely, for the distant strategy, initial quantization is performed conservatively, as weights furthest from a Fibonacci code word are first encoded. Table 3.1 details the fraction of weights to quantize at each step for random, proximal, and distant FCQ strategies. The proportion of weights quantized per epoch are adjusted depending on the impact the quantization will have on the network. In other words, the early epochs of the proximal strategy moves weights less, so more are quantized in the first epochs, and vice versa for the distant strategy.

3.4 Experimental Setup

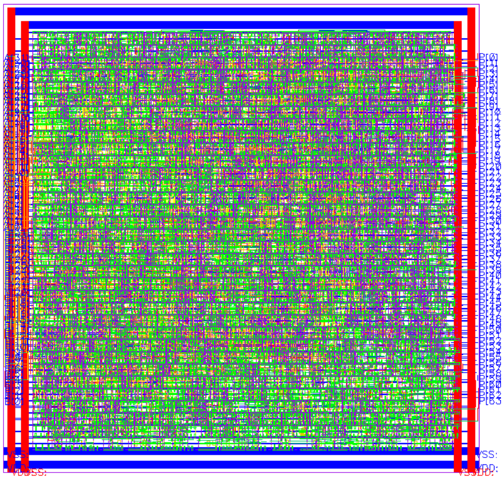


Figure 3.8: Placed and routed 32 bit CAPPIEM in 28nm CMOS.

A hardware assessment of CAPPIEM is performed via implementation in 65nm TSMC CMOS [65] to allow area, Power-Delay-Product (PDP), and MRED comparisons against state-of-the-art AM multipliers [85]. Figure 3.8 illustrates the place and routed implementation of CAPPIEM. A comparison is also made against a subset of AMs from the EvoApprox8b [95] library which are optimized with respect to MRED and power consumption. EvoApprox8b is a set of Pareto-optimal evolved AMs generated by genetic algorithms that have been utilized in previous works to improve neural network efficiency [87]. Finally, CAPPIEM is compared against a 6-bit incrementally quantized network in which we quantize all weight values between 0 and 63, as 6-bit weights enable 64 unique values, comparable to the 55 weight values enabled by FCQ quantization.

To assess the proposed weight quantization and retraining strategy, FCQ-INQ is implemented on the DenseNet-121 [96], Squeezenet 1.0 [97], and ResNet-18 [98] NNs in PyTorch, over the CIFAR-100 database, and analyze accuracy over random, proximal, and distant quantization strategies against standard 8-bit quantization. OneShot FCQ is also performed, in which all weights are quantized in one go. A hyperparameter is defined, namely, Iterative Steps, which contains the fraction of weights to quantize at each step, defined by the values detailed in Table 3.1. The retraining stage commences with a learning rate of 8E-4, decreasing by a factor of 0.2 when the loss value plateaus, and stopping once either the learning rate drops below 1E-6 or 24 epochs are completed. All trainings and inferences are performed with an NVIDIA Tesla T4 GPU.

3.5 Experimental Results and Analysis

3.5.1 Hardware Synthesis and Analysis

As CAPPIEM is implemented by modifying the partial product adder tree, it is best compared against works described in [85], hereafter called the FA Approx library, as well as AMs from the EvoApprox8b library. Area and PDP results are illustrated in Figure 3.9-a. It should be noted that while the EvoApprox library is implemented in 45nm CMOS, the relative area/PDP reductions are still relevant to this analysis. Red points belong to the FA Approx library, which contains two subsets of AMs, one with a fully approximated adder tree, and the other with only LSB approximation. Blue points represent the EvoApprox8b AMs. As can be seen, CAPPIEM provides area/PDP reductions of 73/43%, respectively, with greater area reduction than any FA Approx AM and all but the smallest Evo AM. These reductions are impressive given that only 58% of the FAs were converted into `or` gates, whereas the works in comparison with better reductions consist entirely of approximate FAs.

Figure 3.9-b illustrates the MRED of the AMs in comparison. As can be seen, this work outperforms all fully and most partially approximate AMs from the FA Approx library. Most EvoApprox AMs provide better MRED; however, when taken in conjunction with Figure 3.9-a, this work provides a stronger trade-off between area, PDP, and MRED. Even with a low MRED, retraining is still necessary to recoup lost accuracy. Therefore, the next section analyzes FCQ-INQ retraining features.

3.5.2 Retraining Analysis

In order to demonstrate the importance of FCQ for retraining purposes, inference is performed (representing the forward pass of retraining) via DenseNet-121 on a set of 10000 images divided into 40 batches of 256 images each, while simulating an AM from the FA Approx library via the technique described in [87]. To complete the entire set with AM simulation takes ~20 minutes. In comparison, the same set of inferences without AM simulation takes only 4-5 seconds, or *240-300x* faster. Through FCQ, AM simulation can be avoided, and, hence, networks can be retrained without drastic slowdown.

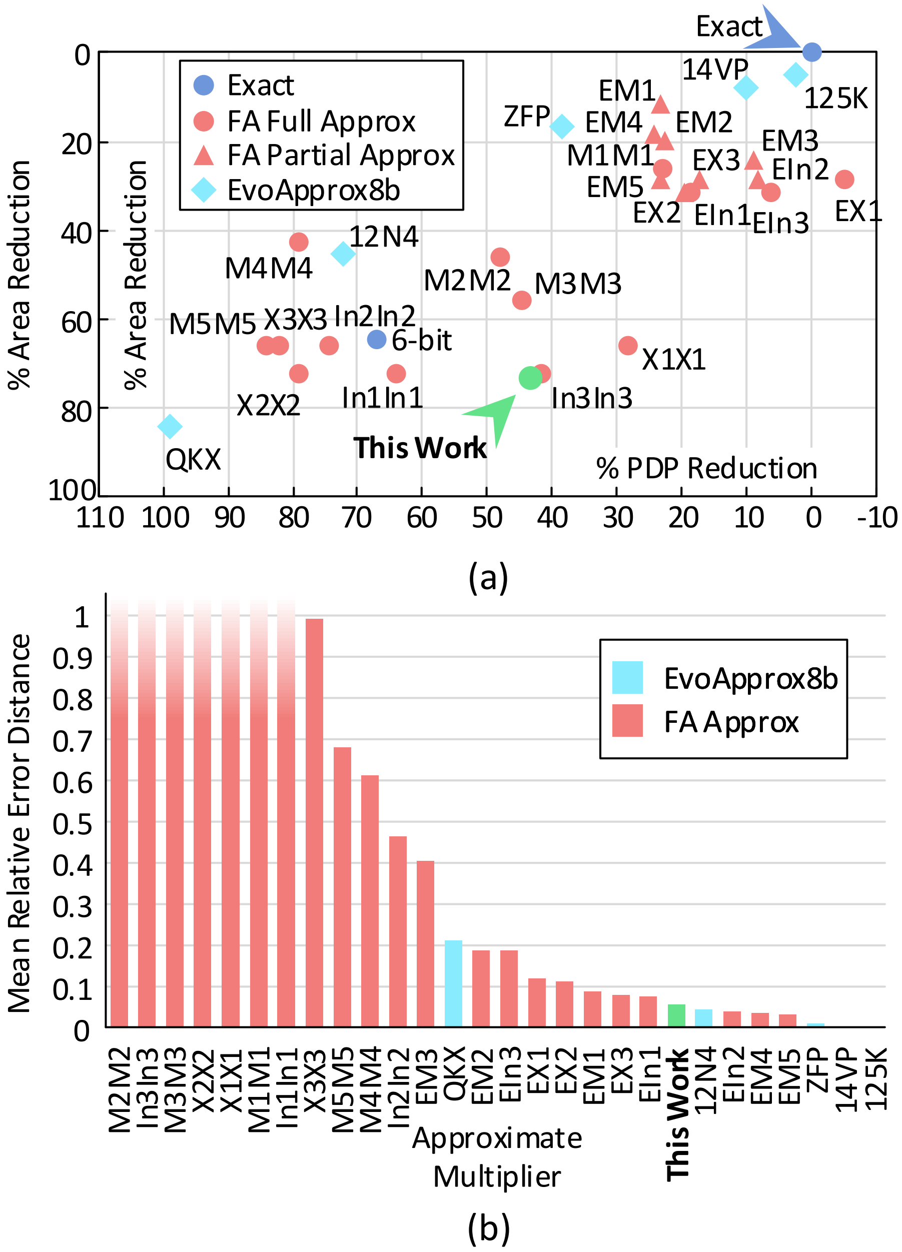


Figure 3.9: (a) Area and Power-Delay-Product (PDP) reduction comparison between this work against other AMs for 8x8 multiplier architectures. (b) Mean Relative Error Distance (MRED) of this work in comparison to other AMs. This work presents high area and PDP reductions in relation to its low MRED.

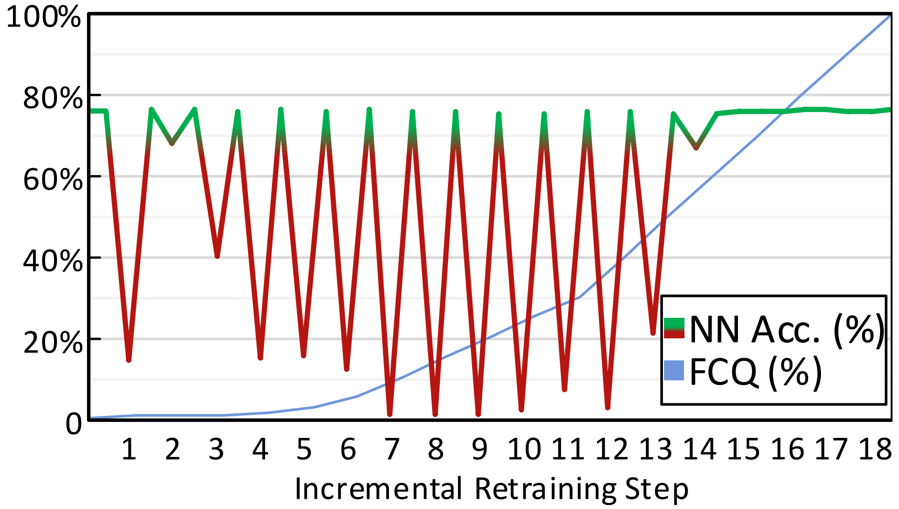


Figure 3.10: DenseNet-121 accuracy during the quantization/retraining process utilizing the distant retraining strategy. Retraining enables to recoup nearly all accuracy loss. Weights are quantized conservatively, as described in Section 3.3.1. The blue line represents the percentage of weights quantized at each step.

Figure 3.10 illustrates the accuracy loss during FCQ and subsequent recovery during retraining for DenseNet-121. It is clear that retraining is necessary to maintain network accuracy, as even FCQ for 0.1% of weight values reduces accuracy by 62%. Via fast retraining, however, almost all lost accuracy is recovered. The ability to fast retrain is nearly unique to the AM presented in this work; all AMs in the FA Approx library, and all but 2 AMs in whole EvoApprox8b library (totaling 46 multipliers) depend on both inputs to ascertain the accuracy of the output, with the 2 Evo AMs that do meet this criterion providing only 4/7% and 0.7/5% area/power reductions.

Table 3.2 details the results of FCQ on the selected benchmarks. As can be seen, 8-bit weight quantization leads to little accuracy degradation. FCQ is then performed using the aforementioned ICQ strategies. Accuracy degradation for the OneShot strategy is predictably poor, as only *bias* and *batchnorm* hyperparameters can be retrained. Next, the random strategy provides reasonable results across all networks. Interestingly, random FCQ performs better than the proximal strategy, due to the aforementioned fact that little

Table 3.2: Accuracy Degradation for quantization strategies after INQ

Network	**ResNet-18**	**DenseNet-121**	**Squeezenet 1.0**
Float Acc. (%)	75.28	77.47	69.24
Quantized Acc. (%)	75.16 (-0.43)	76.37 (-0.10)	68.85 (-0.39)
6b Quant. Acc. (%)	73.78 (-1.5)	67.28 (-1.96)	71.42 (-6.05)
OneShot Acc. (%)	72.42 (-2.86)	64.76 (-12.71)	60.2 (-9.04)
Random Acc. (%)	73.05 (-2.23)	74.62 (-2.85)	66.83 (-2.41)
Proximal Acc. (%)	73.04 (-2.24)	71.64 (-5.83)	65.25 (-3.99)
Distant Acc. (%)	73.54 (-1.74)	76.37 (-1.1)	68.86 (-0.38)

training is done in the early retraining stages. In contrast, the distant strategy recovers nearly all lost accuracy across all networks, resulting in degradations of only 0.4%, 1.1%, and 1.7%. The distant strategy also converges in fewer epochs, resulting in faster training. FCQ also generally outperforms 6-bit quantization as the maximum value to which weights may be quantized to is nearly tripled for FCQ than 6-bit quantization (170 vs. 63), allowing FCQ quantized networks to better capture the variance of the original network.

3.5.3 Comparison to the State-of-the-Art

As the multiplication operation consumes a sizable portion of the total energy cost of inference [87], significant research has been performed to implement approximate multiplication for NNs. Several works [86, 87] utilize genetic algorithms to explore the multiplier design space and simulate the impact of approximation on each layer and neuron of the NN. These works utilize non-uniform AM architectures across the network as well as in some examples mixing AM architectures within a network, demonstrating impressive energy, area, and delay values for the network and dataset but eliminating flexibility to other networks and datasets when implemented in hardware. Many works also require retraining during or after insertion of AMs [86, 88]. Such retraining requires that the approximate multiplier be simulated, precluding the use of hardware optimizations for vectorized multiplication and increasing inference time to hours or days for deeper networks [87]. Finally, partly as a

Table 3.3: Comparison of this work to other approximate multiplier NNs

Work	Dataset	Retrain/Uni./ Depth	Energy Reduction	Accuracy Loss
ApproxANN [88] 2015	MNIST CIFAR-10	Slow / No / Unspecified	-35% -51%	-0.5% -0.5%
Jiao [99] 2018	MNIST	No / Yes / Low	-48%	-1.0%
ALWANN [87] 2019	CIFAR-10	No / No / High	-30%	-0.6% -0.9% -1.7%
This work	CIFAR-100	Fast / Yes / High	-39%	-0.4% -1.1% -1.7%

result of the prohibitive cost of retraining, most works thus far have targeted simpler datasets, such as MNIST, CIFAR-10, or SVHN.

Therefore, this work differs from the majority of AM based NNs in 3 aspects:
1) the CAPPIEM architecture guarantees exact outputs when weight values are properly quantized. This means that it is not necessary to simulate the AM, which leads to the second benefit of this work,
2) network retraining can be performed with hardware optimizations via standard vectorized multiplication. This enables for much deeper approximated networks than has been demonstrated in the SotA, as well as fast exploration of the hyperparameter design space, while still providing large area and power reductions. Finally,
3) the design is applied uniformly across the network, and has thus been demonstrated as generalizable to networks of various depths and feature types, such as Squeezenet's Fire modules and ResNet's residual functions.

Table 3.3 compares this work with other state-of-the-art works across various features, such as dataset, necessity of retraining, uniformity, and network depth. As can be seen, the presented work demonstrates SotA accuracy while maintaining hardware uniformity across the network. While incremental retraining is utilized, it is performed with hardware acceleration, enabling approximation even in deep networks.

3.6 Conclusion

In this chapter, I have presented a hardware/software co-design solution for reducing the area, power, and delay costs of NN multiplications. When implemented as a standalone AM, CAPPIEM replaces over half of a multiplier's full adders with `or` gates, reducing area and PDP by 73/43%. When implemented in BLADE, CAPPIEM halves multiplication latency while only requiring two extra transistors added to the LG periphery. CAPPIEM also has a unique property in that it performs exact multiplications when one input is a Fibonacci code word. I exploit this characteristic by incrementally Fibonacci quantizing and retraining NN weights. The FCQ methodology reduces inference runtime during retraining by 240-300x, while the selected benchmarks, Squeezenet 1.0, DenseNet-121, and ResNet-18, incur very small accuracy losses of 0.4/1.1/1.7% for the CIFAR-100 dataset, while still benefiting from approximate multiplication.

4

Improving Capacity with Hybrid Caches

WHILE the previous chapter described a method for reducing BLADE's latency through approximate multiplication, this chapter focuses on improving the memory capacity available to the application at runtime. Applications in general, and NNs in particular, have become increasingly memory hungry as their levels of accuracy are improve. In an effort to enable NNs on as many devices as possible, many optimizations to reduce NN memory and compute overhead have been proposed, such as quantization, pruning, and custom layers [94, 97, 100]. Even so, the memory footprint of "small" NNs often still measure in the order of MBs [45]; therefore, memory enhancements that exploit the invariant nature of these weights can continue to improve NN performance on area-restricted devices.

In this regard, Figure 4.1 displays the read access to write access (read/write) ratios of the memory blocks that account for 98% of total inference-time read accesses for the SqueezeNet NN [97]. As can be seen, the memory blocks with the highest read/write ratio contain weight values, as these blocks are only written during line fills from lower memory levels during inference, never from the processor. Weight values also account for almost 40% of all memory accesses performed at runtime. It can be inferred that improving processor read access to these weight values will result in overall application performance gain.

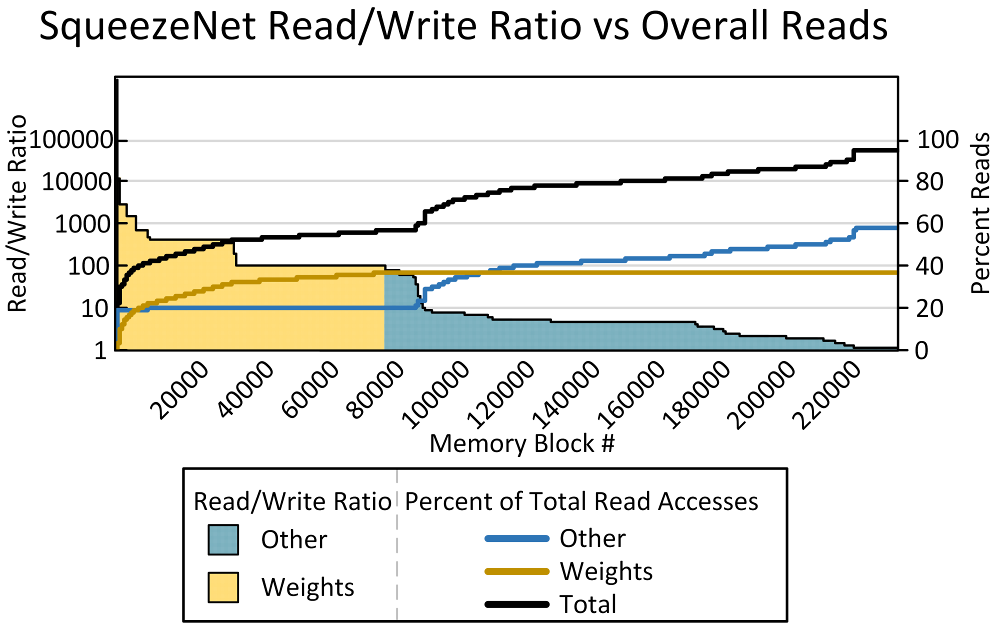


Figure 4.1: Read/write access ratios in relation to total read accesses. Weight access accounts for nearly 40% of reads.

While it would be possible to simply increase SRAM subarray count or size, another option is emerging Non-Volatile Memories (eNVRAMs). eNVRAMs are being exploited for their unique iMC capabilities, which can be integrated with BLADE. In this context, Hybrid Caches (HCs), consisting of SRAM and eNVRAMs, can be used to accelerate NNs. eNVRAM's low area footprint and leakage energy enable more efficient execution of memory intense algorithms by increasing cache capacity with little area overhead, while simultaneously reducing power consumption. However, eNVRAMs also incur a high write energy cost and have limited endurance. It is therefore necessary to optimize write strategies to avoid unnecessary writes. Many works have proposed heuristical, predictive placement strategies. In contrast, a deterministic cache allocation strategy enables the utilization of eNVRAM allocated variables to choose which values are written to eNVRAM and avoid unnecessary transfer between SRAM and eNVRAM, thus providing maximum cache usage efficiency. In the case of NNs, invariant weight values are an excellent candidate for eNVRAM storage.

To this end, I present a HW/SW Stack for Hybrid Caches (SHyCache), consisting of a HC architecture and deterministic cache allocation strategy, supported by a programming model reminiscent of those utilized to enable GPU computation, illustrated in Figure 4.2-a. SHyCache enables precise control over data placement within the cache, and is compatible with heuristical hybrid cache strategies. Then, I explore the HC design space by considering various eNVRAM/SRAM HC ratios, and benchmark SHyCache in gem5-X on a range of NNs of varying computational complexity and memory footprint. I also demonstrate the integration of BLADE into SHyCache, enabling iSC operations to be performed between weights stored in eNVRAM and activations stored in SRAM via a memory bridge and pseudo-local groups.

The contributions of this chapter are as follows:

- I introduce SHyCache, an HC architecture with a deterministic allocation strategy allowing for precise data allocation within an HC. This strategy is compatible with other hybrid cache allocation strategies.
- I develop a programming model with a C++ support library allowing easy integration of SHyCache support into any existing application.
- I implement SHyCache in the gem5-X architectural simulator and explore the HC design space to optimize for performance, power, and endurance, demonstrating performance gains of 1.7/1.4/1.3x and power consumption reductions of 5.1/5.2/5.4x for the Inception v4, ResNet-50, and SqueezeNet 1.0 NNs, respectively.
- I propose a method for co-implementing BLADE and SHyCache to gain the advantages of both iSC and eNVRAM in one system.

The rest of this chapter is organized as follows. Section 4.1 explores related state-of-the-art work. Section 4.2 details SHyCache's HC architecture. Section 4.3 details SHyCache's programming model and support library and discusses tandem implementation with other allocation strategies. Section 4.4 details the benchmarking methodology, while Section 4.5 discusses results. Section 4.6 describes how BLADE and SHyCache can be co-implemented. Finally, Section 4.7 concludes the chapter.

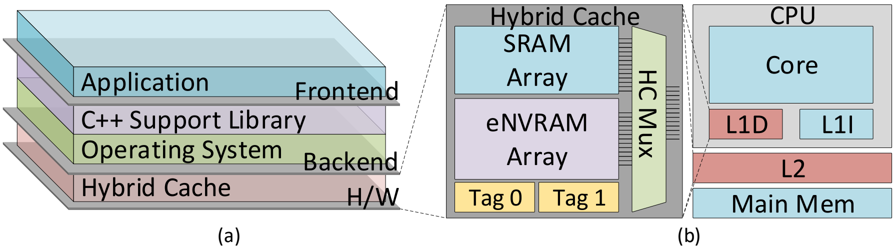


Figure 4.2: SHyCache is a HW/SW stack (a) that enables efficient use of a hybrid cache (b).

4.1 Related Work

4.1.1 Resistive Random Access Memory

Emerging nonvolatile memories, including phase change [101], resistive [102] and spin-torque transfer [103] memories, have recently gained popularity thanks to their small size, up to 4x smaller than 6T SRAM cells [104], and low leakage energy due to their nonvolatility. However, eNVRAM also suffers from long/high-energy write operations, and low endurance due to the underlying physics of the technology. In order to efficiently utilize eNVRAM within an architecture, eNVRAM-specific optimizations must be implemented to magnify their advantages while mitigating or masking drawbacks.

4.1.2 Hybrid Cache Design and Allocation Strategies

One implementation of eNVRAM within the memory hierarchy involves placement alongside standard SRAM cache arrays, creating a Hybrid Cache (HC) hierarchy, as illustrated in Figure 4.2-b. This architecture increases cache capacity while also reducing power consumption [105]. However, HCs also inherit eNVRAM's disadvantages as described above. Further, a naive HC implementation may magnify these disadvantages, as the frequency of cache writes, and therefore cache lifetime, is highly variant depending on the application [106], as well as reducing performance even while not in use due to slower access time. Many works have therefore proposed memory management strategies [107] for allocating blocks in either SRAM or eNVRAM depending on a variety of factors. The majority of these strategies are heuristic [106, 108, 109] or compiler based [110, 111]. In contrast, this chapter presents an application driven allocation strategy which obviates the need for heuristics and takes advantage of cases in which an application's data is constant, such as neural networks.

4.1.3 Neural Networks

Neural Networks (NNs) are a class of applications that process inputs through the use of consecutive compute layers and return an output, for example, the class of the input. Each hidden layer consists of one or more "neurons" of various function. The two most widely used neuron layers are the fully connected and convolutional layer. Both layers perform multiply-and-accumulate operations between the outputs of the previous layer and an array of previously trained weights. These layers require a massive number of weight values; the classical Alexnet NN utilizes 3.78M weights (144MB for floating point weights) in its first fully connected layer [112]. Convolutional layers reduce memory footprint by using small (ex. 3x3) weight kernels that are convolved with the layer input. While convolutional layers greatly reduce the NN's memory footprint, they are generally still large in an absolute sense; for example, the SE-ResNeXt-50 NN achieves the highest Top-1 and Top-5% accuracy on the ImageNet-1k database at a low operational complexity, yet still contains over 10MB of weights [45]. Managing such large quantities of weights is imperative for efficient NN execution.

4.2 Hybrid Cache Architectural Design

SHyCache's hybrid cache consists of arrays of two memory types, one being standard 6T SRAM based memory and the other a flavor of eNVRAM, as illustrated in Figure 4.2-b. Each bitcell array is indexed by a separate tag array. The combined area of the tag array memory macros is equivalent to a single tag array of an equivalently sized monolithic cache memory, plus overhead for tag array periphery. As SHyCache's data placement strategy is deterministic, as described in Section 4.3, only one data/tag array needs be accessed per read/write, reducing power consumption in comparison to heuristic strategies that must check both arrays for the data as its location is not known beforehand. In regards to cache access latency, it is important to note that, as only either the SRAM or eNVRAM is accessed, SHyCache's allocation strategy does not impact access latency of programs not utilizing the eNVRAM, i.e. the system kernel, and thus does not impact standard system performance. This is not necessarily the case if other heuristic or compiler-based allocation strategies are implemented alongside SHyCache's allocation strategy, as discussed in Section 4.3.3.

As illustrated in Figure 4.2-b, HC configurations at both the L1 and L2 levels are considered. An inclusive cache policy is utilized for reasons explained in Section 4.5. The L1 cache utilizes parallel tag/data access to reduce access time as described in Section 2.5, while the L2 uses sequential tag/data access to reduce power consumption.

4.3 Integrating SHyCache's Programming Model into Neural Network Frameworks

Several characteristics of NN weights enable NNs to be accelerated by HCs. The first is that, as previously mentioned, most NNs that achieve >80% Top-5% accuracy utilize large amounts (in the order of MBs) of weights. Second, weight values are calculated at training time and not modified during inference. Finally, fully connected and convolutional layers result in spatially local data accesses. These characteristics make eNVRAM suitable for storing NN weights. High eNVRAM bitcell density allows more weights to be stored without the need for eviction, while the long write latency of eNVRAM is mitigated by the read-only nature of weights.

4.3.1 Enabling HC allocation at the Operating System Level

Most previous HC works utilize heuristic strategies to allocate data either in the SRAM or eNVRAM bitcell arrays depending on various factors. In contrast, because the location and value of NN weight values are deterministic, no heuristic strategy is necessary for weight allocation in SHyCache. This is accomplished at the system level by reserving a portion of memory at operating system startup that can be mapped by an application in the same manner that peripherals can be mapped and accessed by user applications, similar to the strategy detailed in Section 2.5.1. When variables allocated to the memory

```
using namespace SHyCache;
void loadWeights(string weightsFile, size_t len) {
   // Declare var to be stored to eNVRAM portion of
     cache. Allocation handled by helper library.
   float32_nv *weightsPtr = new float32_nv[len];
   // Open file containing pre-calculated weights.
   ifstream wIn(weightsFile);
   // Store weights to previously allocated memory.
   wIn.read((char *)weightsPtr,len);
   //...Perform inference...
   // Clean up
   delete weightsPtr;
}
```

Listing 4.1: Allocating the hybrid cache is done by allocating the variable pointer within the memory mapped region reserved for eNVRAM.

range reserved for eNVRAM caching are fetched into the cache hierarchy, an address predecoder analyzes the MSBs of the incoming address. Addresses within the reserved memory region will be automatically cached in eNVRAM array when accessed. Such a strategy does not require any compiler modification and minimal application modification. Architectural modifications will depend on the nature of the architectures virtual-physical memory address translation. If the reserved memory is virtual, when address translation occurs the processor can tag the memory access with a bit to indicate if it is a standard or eNVRAM memory access before passing the access to the cache hierarchy. If the reserved memory is physical, or there is no virtual-physical translation, for example in embedded systems that use tightly coupled memory [113], the type of memory access will be attained as a byproduct of the address decoding that occurs during cache access, hence, no modification to the processor architecture is necessary.

4.3.2 Enabling HC allocation at the Application Level

At the application level, the programmer utilizes SHyCache's C++ data types to instantiate variables that will be allocated to the eNVRAM, as seen in the example function in Listing 4.1. The support library then facilitates the allocation of variables to the eNVRAM memory region without further programmer intervention by allocating the variables to the memory mapped region described in Section 4.3.1.

Current NN frameworks, such as Tensorflow, Caffe, and ACL perform several preprocessing stages upon weights before storing them in their final tensor, after which this tensor is not modified during inference. Framework extension to support HCs consists therefore of redirecting the output of the final preprocessing stage to store weight values in a tensor stored in the eNVRAM cache, resulting in no extra data movement overhead. In this chapter, I extend ACL with SHyCache's C++ support library, however such extensions could be applied to any of the aforementioned frameworks to enable HC support. It should be noted that the use of a support library obviates the need for any language compiler modifications, simplifying the deployment process.

4.3.3 Co-Implementing SHyCache with Other HC Allocation Strategies

Many SotA works use heuristic methods for moving data between the SRAM and eNVRAM memory domains depending on access patterns [106]. Other works use compiler driven strategies that analyze applications at compile time to choose the best domain for declared variables [110]. One advantage of SHyCache is that it does not preclude the use of these HC allocation strategies. Such strategies can be implemented in tandem by excluding the memory region utilized by SHyCache from the data migration scheme. Even a heuristic allocation strategy with oracle prediction abilities would benefit from SHyCache, as, in order to maintain fast access times, the tag array (and data array in the case of simultaneous tag/data access) of both the SRAM and eNVRAM domains of the HC cache must be accessed simultaneously, as the location of the data is unknown prior to access. On the other hand, SHyCache determines the location of the data at compile time, and the address decoding process routes data access to only the portion of cache in which the data is located, reducing power consumption.

4.4 Experimental Setup

To assess SHyCache's application level performance, gem5-X is extended to support HC caches. three NNs of differing computational complexity and memory footprint are benchmarked, and their performance, power, and endurance trends across a range of HC geometries are analyzed.

Table 4.1: Simulator Parameters

Processor	2GHz, 4 stage pipeline, ARMv8 ISA in-order core, 7 entry LSQ
Co-processor	NEON, 128-bit registers, 16 parallel 8-bit operations
L1-I Cache	32kB, 4-way, 2 cycle access
L1-D Cache	32kB, 4-way, 2 cycle access
L2 Cache	1024/0kB SRAM, 0/4096kB STT-MRAM mostly-inclusive, 16-way, 20 cycle access
STT-MRAM Write Time	50ns [103]
Memory	DDR3 2133MHz, 4GB

4.4.1 gem5-X Simulator Parameters and Hybrid Cache Access Latency Simulation

An ARMv8 A53 in-order core is emulated by calibrating gem5-X with the simulation parameters illustrated in Table 4.1, and simulating an Ubuntu 18.04 LTS software environment. CPU and interconnect power statistics are extracted via the McPAT power estimation framework [55]. SRAM timing and power values are extracted from an implemented subarray in 28nm using TSMC's high performance technology PDK [114], as described in Section 2.4.1. eNVRAM power values are drawn from literature, with STT-MRAM [103] considered for this chapter, however the allocation strategy is technology independent. In order to illustrate SHyCache's performance and power trends, performance and power statistics are extracted across multiple HC hierarchies, in addition to SRAM-only baseline simulations. Hybrid cache geometries are defined by assuming a 4x area ratio between SRAM and eNVRAM bitcell arrays [104], and then sweeping eNVRAM capacity between 0-128kB and 0-4096kB for the L1/L2 caches, respectively, while maintaining an equivalent area footprint.

In order to accurately simulate HC access, SRAM and STT-MRAM access latency is defined in cycles, as documented in Table 4.1. This access latency represents the time to access a cache block through the decoding logic and H-tree, and is pipelined in this implementation, allowing consecutive cache accesses to overlap without blocking. Additionally, STT-MRAM write latency includes an additional write time measured in *ns*, representing the time taken to write a line of data to STT-MRAM. During this time, the subarrays

Table 4.2: Neural Network Benchmark Parameters

Benchmark	# Layers	# Parameters	Weight Memory Footprint (MB)
Inception v4	27	41.1M	156.8
ResNet-50	50	23.5M	89.6
SqueezeNet v1.0	18	1.25M	4.76

being written to cannot be accessed; therefore, this time is not pipelined and subsequent accesses to busy subarrays are blocked. To mitigate this effect, a buffer is implemented between the data bus and the eNVRAM array that stores the latest access to the eNVRAM array and enables coalescing of consecutive writes before the buffer is written to the eNVRAM array. As SHyCache is deterministic in that only the SRAM or eNVRAM portions of the memory need to be accessed for any given cache block, this added latency is not present in standard SRAM accesses, and hence does not impact system performance in cases where the eNVRAM is not accessed.

4.4.2 Neural Network Benchmarks

In order to benchmark SHyCache, three modern NNs of differing sizes are implemented in ACL [59], namely, Inception v4 [115], ResNet-50 [98] and SqueezeNet v1.0 [97], whose parameters are outlined in Table 4.2. These networks enable benchmarking of SHyCache under a wide range of network complexities and memory footprints. All weights and inputs are in floating point, and input batch sizes are set to one.

4.5 Experimental Results and HC Optimization Analysis

SHyCache's experimental results reveal trends in relation to power consumption, runtime performance, and eNVRAM endurance. Analyzing these metrics enables the architect to optimize the HC hierarchy for the system's expected use case.

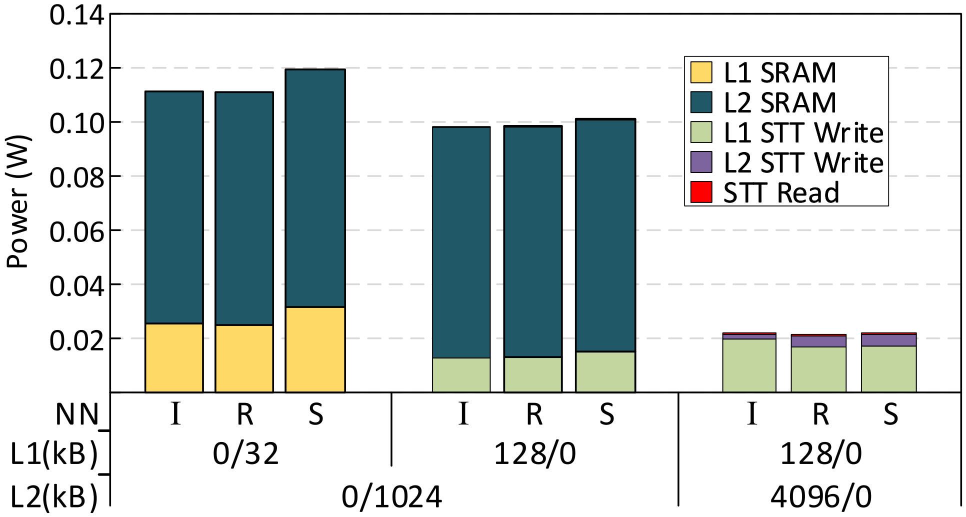


Figure 4.3: Power consumption of all-SRAM, SRAM+eNVRAM, and all-eNVRAM caches for Inception (I), ResNet (R) and SqueezeNet (S) NNs.

4.5.1 Power Results

First, SHyCache's implications on power consumption are considered. Figure 4.3 provides an in-depth breakdown of the power consumption of pure SRAM, pure L1 STT-MRAM, and pure L1/L2 STT-MRAM cache hierarchies. As can be seen, while L1/L2 STT-MRAM write power is substantial, eliminating the energy-leaking SRAM caches provides an excellent reduction in power consumption. STT-MRAM read energy is on par with SRAM read energy, and is too low to be visible in Figure 4.3 when compared to the static energy consumption of the SRAM caches. Figure 4.4-b summarizes the results of the HC design space, from which two trends can be drawn. First, regardless of the L2 cache, a spike in power reduction is seen at a 128kB pure SRAM cache. A slight upward trend in power reduction can also be seen as the STT-MRAM/SRAM ratio of the L2 cache increases, until a sudden jump at a pure STT-MRAM cache. This is because in a pure STT-MRAM cache the power-hungry SRAM bitcell array is replaced with a low leakage STT-MRAM bitcell array. A similar, more pronounced power reduction occurs when replacing the L2 SRAM array entirely with STT-MRAM. Overall, a maximum possible power reduction of 5.1/5.2/5.4x is achieved for Inception/ResNet/SqueezeNet, respectively.

4.5.2 Performance Results

Next, the impact of SHyCache on NN runtime is considered. This is accomplished by performing inference with a batch size of one for the three NNs, normalizing the results to pure SRAM cache hierarchies. The No-L2 portion is normalized to a pure SRAM cache of 32kB, while all other portions are normalized to a 32/1024kB L1/L2 pure SRAM cache hierarchy. Performance as a function of HC architecture is illustrated in Figure 4.4-a

As can be seen, runtime acceleration varies widely across cache geometries. On one hand, if solely the L1 cache is considered in Figure 4.4-a, performance gains of up to 1.31/1.09/1.03x are measured for Inception/ResNet/SqueezeNet, respectively, as the STT-MRAM/SRAM HC ratio is increased up to 64kB/16kB. However, increasing the size of the STT-MRAM array past this point degrades performance, as less SRAM space remains for the remainder of the application. Generally, a 128kB pure STT-MRAM L1 cache results in a steep decrease in performance as all memory accesses, including those with low read/write ratios, are relegated to STT-MRAM. It should also be noted that performance gain attributable to L1 STT-MRAM decreases as computational complexity and memory footprint increases, as the tiny L1 cache becomes insignificant in comparison to the size of the weights, as seen by the decreasing gains for the deeper ResNet and Inception NNs in relation to SqueezeNet.

On the other hand, if the L2 is also considered a very different trend develops. Increasing the HC ratio consistently improves performance for all NN benchmarks, up to a pure eNVRAM array of 4096kB. The larger cache size results in fewer weight evictions, and the mostly-inclusive cache policy mitigates the effects of constant L1 evictions. Additionally, having such a large ratio between L2 and L1 STT-MRAM capacity (64 in the case of a 64kB L1 and 4096kB L2), reduces the negative effects of data repetition that results from an inclusive cache policy. Overall, maximum possible performance gains of 1.7/1.4/1.3x are achieved for Inception/ResNet/SqueezeNet, respectively, when normalized against pure SRAM L1/L2 cache hierarchies.

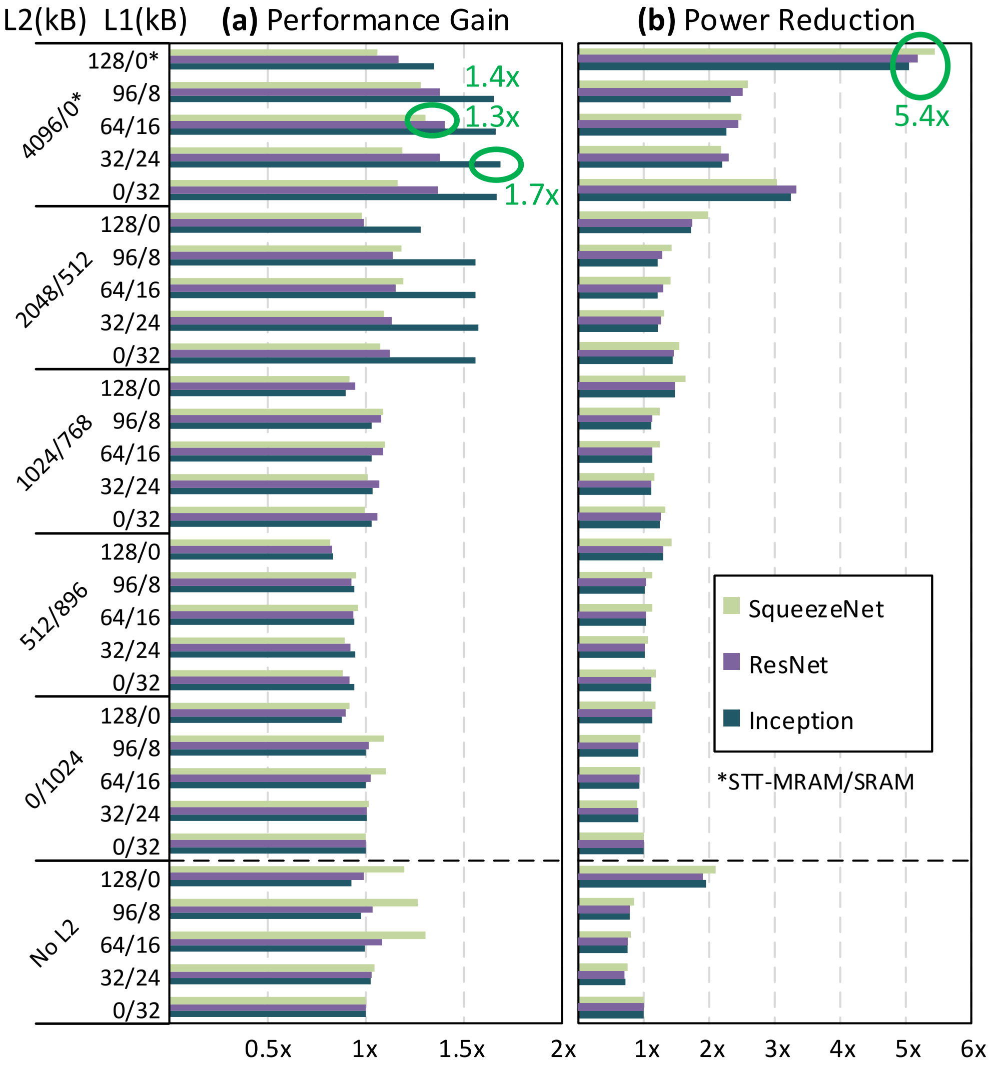


Figure 4.4: (a) Performance gain and (b) power reduction across the HC design space. HC label format is STT-MRAM/SRAM Capacity (kB.)

4.5.3 Endurance Results

Lastly, analysis is made on the number of bitflips that occur within the STT-MRAM array at different cache geometries. eNVRAM life expectancy is tied to its endurance with respect to bitcell value flips, or bitflips. This is measured by counting every 1→0/0→1 flip during writes to the STT-MRAM arrays. As eNVRAM technologies have significantly lower endurance compared to CMOS-based memories, it is imperative to consider bitflip frequency of any architecture utilizing eNVRAM.

Figure 4.5-a illustrates the STT-MRAM bitflip count at all L1 HC geometries with no L2. Consistent with the performance results and reasoning presented in Section 4.5.2, bitflip count drops for 64 and 96kB STT-MRAM caches, before increasing again for pure STT-MRAM caches, with the bitflip reduction more pronounced in the smaller SqueezeNet NN.

Meanwhile, Figure 4.5-b presents STT-MRAM bitflip count for all L2 HC geometries with a pure SRAM L1 cache. The first point of note is that the geometry with the highest bitflip count is not a pure STT-MRAM cache, but in fact an HC of 512/896kB. This is consistent with the performance drop seen across all NNs at this geometry in Section 4.5.2, and is a result of cache thrashing due to the small cache size in relation to the number of weights. Bitflip count then drops as the HC ratio increases and less cache blocks are evicted. Finally, at a pure STT-MRAM cache, the bitflip count for the smaller SqueezeNet NN spikes, as the whole application utilizes STT-MRAM. ResNet and Inception's larger weight footprints dilute this effect, as they gain more from keeping weights in-cache.

This chapter considers only overall bitflip count, not flip counts for individual bits. A drop in average flip per bit is observed as cache capacity increases; however, this metric does not account for uneven intra-word flips skewed toward the LSBs. Many works have explored various eNVRAM wear reducing and leveling optimizations to alleviate this skew. These optimizations are out of this thesis's scope, however, and have not been applied; hence, the numbers demonstrated here are worst case values, with room for future optimization.

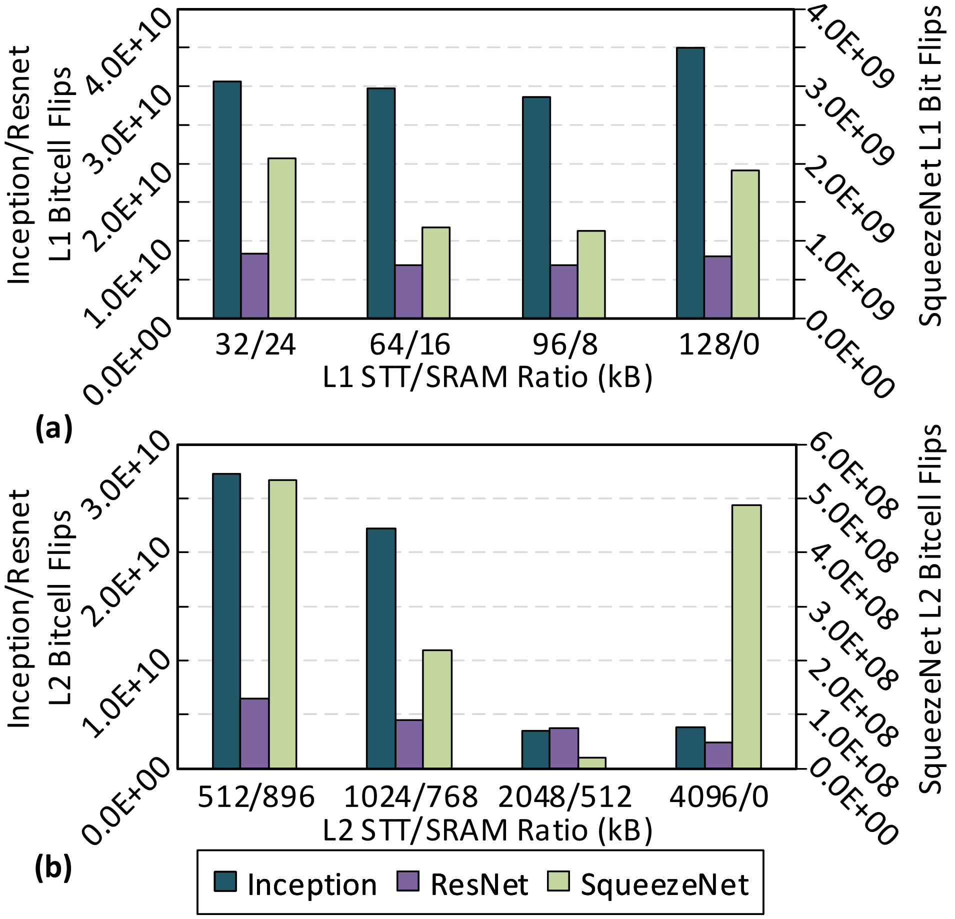


Figure 4.5: STT-MRAM bitcell flips across varying L1 (a) and L2 (b) cache sizes.

4.5.4 Optimizing HCs for Performance, Power, or Endurance

As seen in Sections 4.5.1-3, proper selection of HC geometry for the L1 and L2 caches depends on the system's expected use case. Different geometries optimize either performance, power, or endurance. For example, performance is maximized with a 64/16kB L1 HC cache and a pure STT-MRAM 4096kB L2 cache. However, such a configuration may have a poor endurance when small NNs are the target application. In terms of power, a pure STT-MRAM L1 and L2 provides significant power reduction; however, endurance suffers greatly from such a configuration. From an endurance perspective, the highly active L1 cache accounts for nearly half of all bitflips during inference; a good trade-off between performance, power, and endurance, therefore, may be a pure SRAM L1 with a 2048/512kB L2 HC cache. This architecture provides performance and power improvements of 1.6/1.1/1.1x and 1.5/1.5/1.5x, respectively, while incurring the lowest bitflip count of any architecture.

4.6 Integration with BLADE

SHyCache is also envisaged to be integratable with BLADE. The advantages of such a configuration would be to combine the larger capacity of SHyCache with the SIMD capabilities of BLADE. In order to integrate the two domains, the utilization of pseudo-local groups is proposed.

4.6.1 Pseudo-Local Group Integration

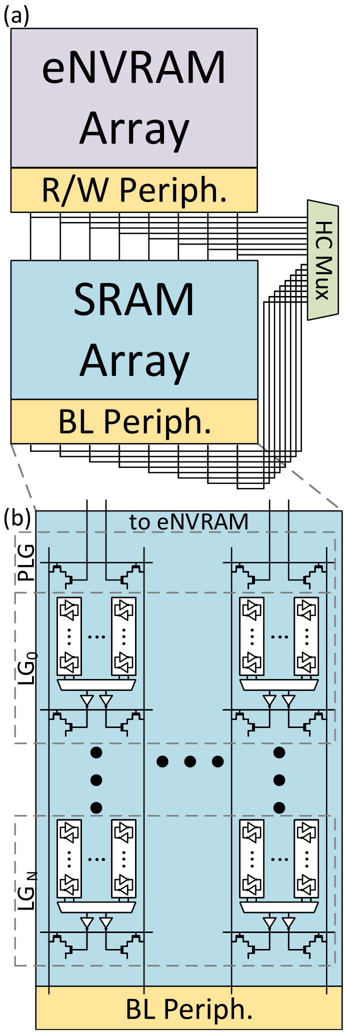


Figure 4.6: a) SHyCache + BLADE architecture with eNVRAM output routed to SRAM subarray.
b) eNVRAM is connected to GBL pairs via PLGs.

In Chapter 2, local groups were introduced as a method for isolating bitcells from each other via read periphery. An interesting aspect of this is that the internal circuitry of the local group is opaque to the global bitline. This means that basically any type of memory or logic can be connected to the read periphery so long as the final output consists of a single bit and its inverse. This can be exploited in the case of SHyCache by routing the output lines from an eNVRAM subarray to a so-called Pseudo-Local Group (PLG) in an SRAM subarray, as illustrated in Figure 4.6-a. This PLG consists of read ports attached to the GBL pairs of the SRAM subarray, enabling operations between the eNVRAM and SRAM subarrays, as illustrated in Figure 4.6-b. This strategy necessitates having an equal number of subarrays of equal width in each domain. The eNVRAM wordline decoder may also need to be modified to allow enabling of a wordline in each subarray to take full advantage of BLADE's SIMD operations.

4.6.2 SHyCache/BLADE Operand Locality

The utilization of PLGs to co-implement SHyCache and BLADE obviates some of the data locality constraints described in Section 2.5.1. Namely, since no SRAM set can possibly share a local group with a set in the eNVRAM, the restriction on the MSBs of the set bits is lifted. However, it is still necessary that the operands occupy matching bitline columns in sister subarrays. Therefore, it is still necessary that the offset bits and n set LSBs match, where n equals $log_2\left(Val_{geo}\right)$ with Val_{geo} defined as in Section 2.5.1.

4.7 Conclusion

This chapter presented SHyCache, a hybrid cache with a deterministic allocation strategy and supporting programming model designed to improve NN runtime while reducing power consumption. SHyCache enables NN frameworks to explicitly allocate weight values to the eNVRAM cache, eliminating data transitions between SRAM and eNVRAM arrays and providing maximal cache efficiency. In this chapter, I explained how SHyCache can be implemented at the system and application level and in tandem with other HC allocation strategies, I have developed a C++ support library allowing implementation in current applications, and I benchmarked SHyCache on three NN applications of varying computational complexity and memory footprint. Analysis of experimental results have demonstrated maximum performance gains of 1.7/1.4/1.3x and power consumption reductions of 5.1/5.2/5.4x, for the Inception/ResNet/SqueezeNet benchmarks, respectively. I considered the implications of the demonstrated results for optimizing an architecture based on expected use cases and proposed a middle-ground solution that provides optimal trade-off between performance, power, and endurance. Finally, I proposed a method for co-implementing SHyCache and BLADE, accounting for architectural considerations and operand locality constraints. Integration benchmarks shall be considered in future work.

5

Conclusion and Future Work

As Moore's law has slowed and Dennard scaling has ended in the last years, innovative methods of continuing to scale runtime, energy consumption and area footprint are being explored at every level of the HW/SW stack. Among these, iMC, both in traditional and emerging memory, reduces data movement while increasing the number of simultaneous operations that can be performed, reducing runtime and energy consumption with minimal area overhead. As a relatively new field, the opportunities for further research presented by iMC is manifold, from incremental changes to preexisting architectures to radical new designs that nearly entirely replace the processor.

In this thesis, I introduced BLADE, an iMC accelerator that utilizes a number of novel techniques for reducing area overhead and running at a frequency higher than other SotA works of a similar nature. Further, I have explored approximate computing and hybrid caches as methods for enhancing BLADE by reducing multiplication latency and expanding memory capacity.

5.1 BLADE: A Base for Future iMC Exploration

BLADE as a platform has many benefits that set it apart from other works in the iMC field. First, its low-overhead implementation in commodity SRAM memory means it is easily and reliably testable and can be implemented in the computer and embedded systems of today, as opposed to many works that rely on emerging technologies that are not yet well characterized. It supports many bitwise and arithmetic operations, including multiplication, which is necessary for many important kernels used in modern day algorithms. In contrast to many SotA analog-based works, its digital nature means that it produces the same results as traditional von Neumann architectures, eliminating the need to evaluate the impact of acceleration on application accuracy, a key complexity that faces other approximate solutions. Simultaneously, the innovative use of local groups and the proper in-memory arrangement of operands enables BLADE to run at native memory frequency, up to 3x faster than previous SotA iSC based works.

Not wanting to be constrained to any one level of the HW/SW stack, I have explored and validated the benefits of BLADE from the hardware all the way to application level:

- BLADE has been implemented in 28nm TSMC CMOS technology and validated through a multitude of electrical simulations. Further, BLADE has also been implemented and fabricated in 60nm TSMC CMOS, with validation via electrical simulation and upcoming characterization of the circuit pending return from the fabricator.

- I explored multiple methods of implementing BLADE into a system architecture. Namely, I implemented BLADE in the L1 cache of a CPU, performing architecture exploration via modification of the cache hierarchy, geometry, and capacity. I also fabricated BLADE within the tightly coupled memory found in the PULP platform, with further characterization and application exploration planned in the near future.

- I developed multiple methods for integrating BLADE at the kernel level. One such iteration treats BLADE as a memory mapped accelerator tightly coupled to the memory hierarchy that could be mapped by application developers for use in applications. This method has the advantage of ease of use and familiarity to embedded system programmers. A second iteration utilizes ISA extensions to send operand information as memory read operations that are then decoded by the

BLADE controller. This iteration has the advantage of more permanent future integration with compiler support and optimizations.

- I implemented BLADE in the gem5-X architectural simulator, enabling timing accurate application level simulations for ascertaining the utility of BLADE at the application level. I analyzed applications utilizing both bitwise and arithmetic operations to see how architectural modifications affect functionality. Implementation in gem5-X highlighted challenges not visible at the hardware level, such as data locality, and provided insights into ways BLADE could be enhanced, such as embedded shifting and way-agnostic applications, which were then implemented in later iterations.

With BLADE implemented at the HW, architectural, and SW levels, many avenues of further research are open for continuing to enhance its in-memory computational capabilities.

5.2 Exploring Opportunities for Enhancing BLADE

Continuing my research, I developed various strategies for improving BLADE:

- In an effort to reduce multiplication cycle count, I developed a CArryless Partial Product InExact Multiplier (CAPPIEM) that can be implemented in BLADE at a low area overhead, reducing multiplication latency by 2x. CAPPIEM can also be implemented as a standalone approximate multiplier, reducing area and power-delay-product by 73%/43%, respectively. CAPPIEM has a unique property among other approximate multipliers in that if one operand is a Fibonacci codeword, aka, if it has no consecutive ones in its binary representation, then CAPPIEM produces an exact product value. With this knowledge, I designed a retraining algorithm called Fibonacci Code Quantization, that quantizes neural network weights to Fibonacci values over multiple iterations, retaining network accuracy while eliminating approximate product values during inference. This algorithm improves retraining time by up to 300x, while incurring very small accuracy losses on NN benchmarks.

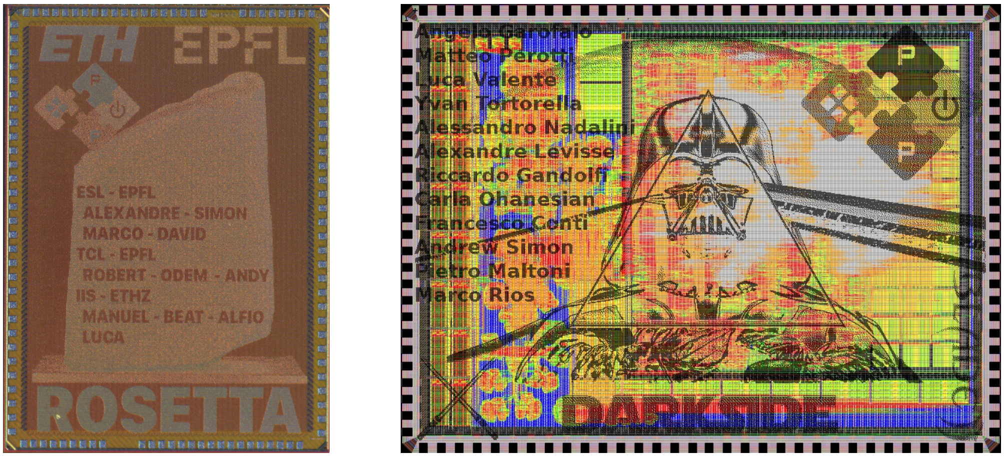
- As emerging non-volatile memories grow in popularity due to their small area footprint, the concept of hybrid caches has also been explored. In this vein, I have implemented a HW/SW Stack for Hybrid Caches (SHyCache), a HW/SW co-designed hybrid cache with deterministic data placement. Utilization of the C++ support library allows the application designer to easily allocate static variables to the eNVRAM domain of SHyCache, improving performance by 1.7/1.4/1.3x and reducing power consumption by 5.1/5.2/5.4x for three deep neural network benchmarks. SHyCache can be further enhanced by BLADE for greater improvements, and could even be extended by one of the many in-eNVRAM computing implementations for further performance gains.

5.3 Future Work

In the future, many improvements and further research could be performed to make BLADE an even more effective accelerator. This is a list of just a few avenues of future work that could be pursued.

5.3.1 Validation of BLADE Fabrication

During my PhD, our team fabricated BLADE in 65nm TSMC CMOS technology on the Rosetta chip, illustrated in Figure 5.1-a, in collaboration with the Integrated Systems Laboratory (IIS) at Swiss Federal Institute of Technology in Zürich (ETHZ). BLADE was implemented in a 32kB, 16 local group SRAM array alongside a RISCY [48] processor core. While the opportunity was exciting and impossible to pass up, a tight time constraint of three months to implement BLADE from the ground up in a new technology node resulted in a faulty circuit. In 2021 we had a second opportunity to fabricate BLADE in IIS's Darkside chip, illustrated in Figure 5.1-b, building on the Rosetta design and integrating further improvements. Darkside has been successfully fabricated and will be characterized in the coming weeks. The Darkside chip will be valuable for validating our previous works' results, and also for performing more complex application level analysis, as Darkside will execute applications orders of magnitude faster than the gem5-X simulator.



(a) Rosetta (b) Darkside

Figure 5.1: Chips fabricated in collaboration with the Integrated Systems Laboratory (IIS) at Swiss Federal Institute of Technology in Zürich (ETHZ).

5.3.2 FPGA Implementation to Reduce Simulation Time

While a physical chip demonstrates that BLADE can be implemented outside of simulation, it of course cannot be utilized for architectural exploration, as its architectural parameters cannot be modified. gem5-X has proven useful as a preliminary test platform for BLADE, but as application benchmarks become more complex, the extremely low execution time has become a barrier to further experimentation. Indeed, running a full deep neural network benchmark, such as ResNet or Inception, takes hours to weeks, depending on the parameters. In order to test full applications, it will be necessary to re-implement BLADE on an FPGA with a softcore processor. Collaboration with IIS-ETHZ and access to their PULP platform which has already been implemented on-FPGA, as well as the implementation of the BLADE controller in HDL for the Rosetta and Darkside projects, put this goal within reach in the near future. FPGA implementation will also decrease development time and reduce the incidence of bugs in future fabrication efforts, as the latest iteration of BLADE will already be available and debugged in HDL.

5.3.3 BLADE Compiler Support

At the current stage of development, two methods for instantiating BLADE at the application level have been described; mapping BLADE as a memory mapped device, and using custom opcodes for sending commands. The second option is the beginning of another research avenue to be explored in relation to BLADE, namely, compiler support and optimization. Currently, application developers are responsible for allocating memory to conform with data locality constraints, and must find any possible optimization themselves. Compilers, on the other hand, are very adept at optimizing applications automatically and reduce the workload on application developers significantly. Particularly, optimizing data access and movement will be paramount to allowing BLADE to be generalized quickly to a wider range of applications. Such support could be implemented in LLVM or Clang, given the ease of modification in comparison to GCC, and would represent the final piece in true full stack support for BLADE.

5.3.4 Further Refinement/Enhancement of BLADE Hardware and Architecture

Finally, the BLADE hardware presents numerous opportunities for improvement. To name just a subset of promising research lines:

- While solvable to a degree at the compiler level, data locality constraints pose a challenge for all types of iMC devices. Reducing these constraints will make application development and deployment significantly easier. As demonstrated in [70], small hardware changes can greatly reduce constraints. While this work succeeded in removing the horizontal data placement limitation imposed by ways, reducing vertical limitations imposed by LGs and separate, non-communicating subarrays poses another challenge. One such solution could be the PLGs described in Chapter 4. These PLGs could be attached to other subarrays, allowing inter-subarray operations, or the incoming data-in wires, allowing lower level memory or the CPU to provide operands as well.

- At the current stage, BLADE has only been implemented in the L1 cache. This was decided upon to avoid the challenge of data coherence between multiple processors, but there is no reason why BLADE could not be implemented at lower levels. Indeed, this would be greatly beneficial for multiple reasons. First, multiple CPUs could share the same BLADE accelerator, mitigating area overhead. Second, more and wider subarrays translate to more operations performed per cycle, increasing throughput. Third, the larger cache size reduces cache eviction when working on kernels with large inputs, such as NN layers with many channels. Finally, data movement is greatly reduced as data is not moved further up the cache hierarchy than necessary. In order to implement such an architecture, BLADE will have to be integrated into the coherency subsystem of the target architecture in much the same way as a DMA. The complexity of such an implementation is not completely trivial, but should be doable by a researcher with a competent understanding of multiprocessor computer architecture.

- Increasing the number of supported operations will improve BLADE's flexibility to perform various kernels. While the current iteration supports bitwise operations, shifting, adding, and multiplication, many other operations could be supported at various degrees of complexity. Implementing `not` is relatively straight forward, opening the door for subtraction, division, and comparison operators. The ReLU activation function used in many neural networks can be implemented by checking the MSB of the value in question. Finally, implementing right-shifting in a manner similar to [70] would enable faster divisions.

- While most operations can be performed in two cycles, multiplication is still a high latency operation. This is mitigated by performing many operations simultaneously; however, further reducing multiplication latency should be a top priority going forward. Enhancements such as the add-forward line presented in Chapter 2 and embedded shifting as presented in [70] demonstrate two methods for reducing latency, and further design space exploration will undoubtedly reveal more possibilities. Implementing BLADE in larger SRAM arrays, e.g., at lower cache levels, will also mitigate multiplication latency.

5.4 List of Publications

- W. A. Simon, V. Ray, A. Levisse, G. Ansaloni, M. Zapater and D. Atienza, "Exact Neural Networks from Inexact Multipliers via Fibonacci Weight Encoding," 58th ACM/IEEE Design Automation Conference (DAC), 2021.
- W. A. Simon, A. Levisse, M. Zapater and D. Atienza, "A Hybrid Cache HW/SW Stack for Optimizing Neural Network Runtime, Power and Endurance," IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC), 2020.
- W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater and D. Atienza, "BLADE: An in-Cache Computing Architecture for Edge Devices," in IEEE Transactions on Computers, 2020.
- M. Rios, W. Simon, A. Levisse, M. Zapater and D. Atienza, "An Associativity-Agnostic in-Cache Computing Architecture Optimized for Multiplication," IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC), 2019.
- W. A. Simon, Y. M. Qureshi, A. Levisse, M. Zapater, and D. Atienza, BLADE: A BitLine Accelerator for Devices on the Edge. In Proceedings of the 2019 on Great Lakes Symposium on VLSI (GLSVLSI '19), 2019.
- W. A. Simon, J. Galicia, A. Levisse, M. Zapater and D. Atienza, "A Fast, Reliable and Wide-Voltage-Range In-Memory Computing Architecture," 56th ACM/IEEE Design Automation Conference (DAC), 2019.

Bibliography

[1] Embedded system market by hardware (mpu, mcu, application-specific integrated circuits, dsp, fpga, and memories), software (middleware, operating systems), system size, functionality, application, region - global forecast to 2025. Technical report, Markets and Markets Research, 2019.

[2] 2019 embedded markets study. Technical report, AspenCore, 2019.

[3] Knud Lasse Lueth, Matt Wopata, Eugenio Pasqua, Misha Rykov, Satyajit Sinha, Sharmila Annaswamy, Philipp Wegner, Marta Simeonova, Nancy Srivastava, Kuan Ngiam, Padraig Scully, and Fernando Bruegge. State of iot q4/2020 & outlook 2021. Technical report, IoT Analytics, 2020.

[4] Blesson Varghese, Eyal de Lara, Aaron Ding, Cheol-Ho Hong, Flavio Bonomi, Schahram Dustdar, Paul Harvey, Peter Hewkin, Weisong Shi, Mark Thiele, and Peter Willis. Revisiting the arguments for edge computing research. *IEEE Internet Computing*, 2021.

[5] The mobile internet phenomena report. Technical report, Sandvine, 2019.

[6] The future is here: iphone x. https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/. 12 2017.

[7] John Backus. Can programming be liberated from the von neumann

style?: A functional style and its algebra of programs. *Commun. ACM*, 1978.

[8] Alexandre Levisse, Marco Rios, W.-A. Simon, P.-E. Gaillardon, and D. Atienza. Functionality enhanced memories for edge-ai embedded systems. In *19th NVMTS*, 2019.

[9] G. Singh, L. Chelini, S. Corda, A. Javed Awan, S. Stuijk, R. Jordans, H. Corporaal, and A. Boonstra. A review of near-memory computing architectures: Opportunities and challenges. In *DSD*, 2018.

[10] J. T. Pawlowski. Hybrid memory cube (hmc). In *HCS 23*, 2011.

[11] Mario Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The mondrian data engine. In *44th ISCA*, 2017.

[12] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *DaMoN 11*, 2015.

[13] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA 42*, 2015.

[14] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *HPDC 23*, 2014.

[15] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *ISCA 43*, 2016.

[16] Maya Gokhale, Scott Lloyd, and Chris Hajas. Near memory data structure rearrangement. In *MEMSYS*, 2015.

[17] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *HPCA*, 2016.

[18] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *HPCA 21*, 2015.

[19] S. F. Yitbarek, T. Yang, R. Das, and T. Austin. Exploring specialized near-memory processing for data intensive operations. In *DATE*, 2016.

[20] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 2020.

[21] Mohsen Imani, Abbas Rahimi, Deqian Kong, Tajana Rosing, and Jan M. Rabaey. Exploring hyperdimensional associative memory. In *HPCA*, 2017.

[22] Seungchul Jung, Hyungwoo Lee, Sungmeen Myung, Hyunsoo Kim, Seung Keun Yoon, Soon-Wan Kwon, Yongmin Ju, Minje Kim, Wooseok Yi, Shinhee Han, et al. A crossbar array of magnetoresistive memory devices for in-memory computing. *Nature*, 2022.

[23] Mustafa Ali, Akhilesh Jaiswal, Sangamesh Kodge, Amogh Agrawal, Indranil Chakraborty, and Kaushik Roy. Imac: In-memory multi-bit multiplication and accumulation in 6t sram array. *Trans Circuits Syst I Regul Pap*, 2020.

[24] Shaizeen Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *HPCA*, 2017.

[25] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache Automaton. *MICRO*, 2017.

[26] Mingu Kang, Sujan K. Gonugondla, Ameya Patil, and Naresh R. Shanbhag. A 481pj/decision 3.4m decision/s multifunctional deep in-memory inference processor using standard 6t SRAM array. *CoRR*, abs/1610.07501, 2016.

[27] Kaya Can Akyel, H. P. Charles, J. Mottin, B. Giraud, G. Suraci, S. Thuries,

and J. P. Noel. DRC2: Dynamically reconfigurable computing circuit based on memory architecture. In *ICRC*, 2016.

[28] ARM. Introducing neon. Technical report, ARM, 2009.

[29] Chris Lomont. Introduction to intel advanced vector extensions. Technical report, Intel, 05 2011.

[30] Supreet Jeloka, N.B. Akesh, D. Sylvester, and D. Blaauw. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6t bit cell enabling logic-in-memory. *JSSC*, 2016.

[31] Supreet Jeloka, N.B. Akesh, D. Sylvester, and D. Blaauw. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6t bit cell enabling logic-in-memory. *JSSC*, 2016.

[32] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy. X-sram: Enabling in-memory boolean computations in cmos static random access memories. *Trans. Circuits Syst. I*, 2018.

[33] W. Khwa, J. Chen, J. Li, X. Si, E. Yang, X. Sun, R. Liu, P. Chen, Q. Li, S. Yu, and M. Chang. A 65nm 4kb algorithm-dependent computing-in-memory sram unit-macro with 2.3ns and 55.8tops/w fully parallel product-sum operation for binary dnn edge processors. In *ISSCC*, 2018.

[34] F. Hsueh, H. Chiu, C. Shen, J. Shieh, Y. Tang, C. Yang, H. Chen, W. Huang, B. Chen, K. Chen, G. Huang, W. Chen, K. Hsu, S. R. Srinivasa, N. Jao, A. Lee, H. Lee, V. Narayanan, K. Wang, M. Chang, and W. Yeh. Tsv-free finfet-based monolithic 3d+-ic with computing-in-memory sram cell for intelligent iot devices. In *IEDM*, 2017.

[35] Srivatsa Srinivasa, Akshay Krishna Ramanathan, Xueqing Li, Wei-Hao Chen, Fu-Kuo Hsueh, Chih-Chao Yang, Chang-Hong Shen, Jia-Min Shieh, Sumeet Gupta, Meng-Fan Marvin Chang, Swaroop Ghosh, Jack Sampson, and Vijaykrishnan Narayanan. A monolithic-3d sram design with enhanced robustness and in-memory computation support. In *ISLPED*, 2018.

[36] William Simon, Juan Galicia, Alexandre Levisse, Marina Zapater, and David Atienza. A fast, reliable and wide-voltage-range in-memory computing architecture. In *DAC*, 2019.

[37] M. Kang, S. K. Gonugondla, A. Patil, and N. R. Shanbhag. A multi-functional in-memory inference processor using a standard 6t sram array. *JSSC*, 2018.

[38] J. Zhang, Z. Wang, and N. Verma. In-memory computation of a machine-learning classifier in a standard 6t sram array. *JSSC*, 2017.

[39] S. K. Gonugondla, M. Kang, and N. Shanbhag. A 42pj/decision 3.12top-s/w robust in-memory machine learning classifier with on-chip training. In *ISSCC*, 2018.

[40] Akhilesh Jaiswal, Indranil Chakraborty, Amogh Agrawal, and Kaushik Roy. 8t SRAM cell as a multi-bit dot product engine for beyond von-neumann computing. *CoRR*, abs/1802.08601, 2018.

[41] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA 45*, 2018.

[42] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester. A compute sram with bit-serial integer/floating-point operations for programmable in-memory vector acceleration. In *ISSCC*, 2019.

[43] William Simon, Yasir Mahmood Qureshi, Alexandre Sébastien Julien Levisse, Marina Zapater Sancho, and David Atienza Alonso. Blade: A bitline accelerator for devices on the edge. *GLSVLSI 29*, 2019.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[45] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 2018.

[46] Elisabetta De Giovanni, Fabio Montagna, Benoôt W. Denkinger, Simone Machetti, Miguel Peón-Quirós, Simone Benatti, Davide Rossi, Luca Benini, and David Atienza. Modular design and optimization of biomedical applications for ultralow power heterogeneous platforms. *TCAD*, 2020.

[47] Rakesh Chadha and Jayaram Bhasker. *An ASIC low power primer: analysis, techniques and specification*. Springer Science & Business Media, 2012.

[48] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley*, 2011.

[49] Robert H Norman. Solid state switching and memory apparatus, U.S. Patent US3562721A, 03 1963.

[50] Virtuoso system design platform. Technical report, Cadence, 2017.

[51] Innovus implementation system. Technical report, Cadence, 2019.

[52] Sigasi studio. https://www.sigasi.com/.

[53] Hdl simulation modelsim. Technical report, Siemens, 2019.

[54] Yasir Mahmood Qureshi, William Andrew Simon, Marina Zapater Sancho, Katzalin Olcoz, and David Atienza Alonso. Gem5-x: A gem5-based system level simulation framework to optimize many-core platforms. *HPC*, 2019.

[55] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *21st HPCA*, 2015.

[56] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6.0: A tool to model large caches. Technical report, Hewlett-Packard, 2009.

[57] Ubuntu 18.04: Bionic beaver. https://wiki.ubuntu.com/BionicBeaver.

09 2018.

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *32nd NeurIPs*. 2019.

[59] A software library for machine learning. "https://developer.arm.com/technologies/compute-library". 2018.

[60] L. Chang, Y. Nakamura, R. K. Montoye, J. Sawada, A. K. Martin, K. Kinoshita, F. H. Gebara, K. B. Agarwal, D. J. Acharyya, W. Haensch, K. Hosokawa, and D. Jamsek. A 5.3ghz 8t-sram with operation down to 0.41v in 65nm cmos. In *VLSI*, 2007.

[61] Qing Dong, S. Jeloka, M. Saligane, Y. Kim, M. Kawaminami, A. Harada, S. Miyoshi, M. Yasuda, D. Blaauw, and D. Sylvester. A 4 + 2t sram for searching and in-memory computing with 0.3-v v_{ddmin}. *JSSC*, 2018.

[62] Reda Boumchedda, J. Noel, B. Giraud, K. C. Akyel, M. Brocard, D. Turgis, and E. Beigne. High-density 4t sram bitcell in 14-nm 3-d coolcube technology exploiting assist techniques. *VLSI*, 2017.

[63] Mahmut E. Sinangil, Hugh Mair, and Anantha P. Chandrakasan. A 28nm high-density 6T SRAM with optimized peripheral-assist circuits for operation down to 0.6V. *ISSCC*, 2011.

[64] M. Schlag and P. Chan. Analysis and Design of CMOS Manchester Adders with Variable Carry-Skip. *TC*, 1990.

[65] 28nm technology. https://www.tsmc.com/english/dedicatedFoundry/technology/28nm.htm. 2011.

[66] M. Chang, C. Chen, T. Chang, C. Shuai, Y. Wang, and H. Yamauchi. A 28nm 256kb 6T-SRAM with 280mV improvement in VMINusing a

dual-split-control assist scheme. *ISSCC*, 2015.

[67] H. Pilo, C. A. Adams, I. Arsovski, R. M. Houle, S. M. Lamphier, M. M. Lee, F. M. Pavlik, S. N. Sambatur, A. Seferagic, R. Wu, and M. I. Younus. A 64Mb SRAM in 22nm SOI technology featuring fine-granularity power gating and low-energy power-supply-partition techniques for 37% leakage reduction. *ISSCC*, 2013.

[68] Taejoong Song, Woojin Rim, Sunghyun Park, Yongho Kim, Giyong Yang, Hoonki Kim, Sanghoon Baek, Jonghoon Jung, Bongjae Kwon, Sungwee Cho, Hyuntaek Jung, Yongjae Choo, and Jaeseung Choi. A 10 nm finfet 128 mb sram with assist adjustment system for power, performance, and area optimization. *ISSCC*, 2017.

[69] M. Kooli, H. Charles, C. Touzet, B. Giraud, and J. Noel. Smart instruction codes for in-memory computing architectures compatible with standard sram interfaces. In *DATE*, 2018.

[70] Marco Rios, William Simon, Alexandre Levisse, Marina Zapater, and David Atienza. An associativity-agnostic in-cache computing architecture optimized for multiplication. In *27th VLSI-SoC*, 2019.

[71] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in computer architecture evaluation. *Computer*, 2003.

[72] Mahmoud Elkhodr, Seyed A. Shahrestani, and Hon Cheung. The internet of things: New interoperability, management and security challenges. *CoRR*, abs/1604.04824, 2016.

[73] Saurabh Singh, P. Sharma, S. Yeon Moon, and J. Hyuk Park. Advanced lightweight encryption algorithms for iot devices: survey, challenges and solutions. *JAIHC*, 2017.

[74] ARM. Arm cortex-a53 mpcore processor cryptography extension. Technical report, ARM, 12 2014.

[75] Morris J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *FIPS*, 2015.

[76] Marko Viitanen, A. Koivula, A. Lemmetti, A. Ylä-Outinen, J. Vanne, and T. Hämäläinen. Kvazaar: Open-source hevc/h.265 encoder. In *ACMMM 24*, 2016.

[77] Song Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *ISCA 43*, 2016.

[78] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

[79] Vinayak Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *CVPR*, 2014.

[80] ARM. Arm versatile express juno r2 development platform. Technical report, ARM, 11 2015.

[81] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *HPCA 21*, 2015.

[82] Kevin Krewell. Cortex-a53 is arm's next little thing. Technical report, The Linley Group, 11 2012.

[83] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *24th VLSID*, 2011.

[84] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo. Low-power high-speed multiplier for error-tolerant application. In *EDSSC*, 2010.

[85] Mahmoud Masadeh, Osman Hasan, and Sofiene Tahar. Comparative study of approximate multipliers. In *GLSVLSI*, 2018.

[86] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. Axnn: Energy-efficient neuromorphic systems using approximate computing. In *ISLPED*, 2014.

[87] Vojtech Mrazek, Zdenek Vasicek, Lukas Sekanina, Muhammad Hanif,

and Muhammad Shafique. Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining. In *ICCAD*, 2019.

[88] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. Approxann: An approximate computing framework for artificial neural network. In *DATE*, 2015.

[89] F. Ponzina, M. Peon, A. Burg, and D. Atienza. E2cnns: Ensembles of convolutional neural networks to improve robustness against memory errors in edge-computing devices. *IEEE TC*, 2021.

[90] L. Ni and Kai Hwang. Vector-reduction techniques for arithmetic pipelines. *IEEE Transactions on Computers*, 1985.

[91] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *ICCAD*, 2016.

[92] Aviezri S. Fraenkel and Shmuel T. Kleinb. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 1996.

[93] Benoit Jacob, Peter Warden, Guney Murat Efe, and Gouicem Mourad. gemmlowp: a small self-contained low-precision gemm library. https://github.com/google/gemmlowp. 2015.

[94] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, 2017.

[95] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *DATE*, 2017.

[96] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[97] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *CoRR*, 2016.

[98] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[99] X. Jiao, V. Akhlaghi, Y. Jiang, and R. K. Gupta. Energy-efficient neural networks using approximate computation reuse. In *DATE*, 2018.

[100] Son Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *CoRR*, 2015.

[101] G. W. Burr, M. J. Brightsky, A. Sebastian, H. Cheng, J. Wu, S. Kim, N. E. Sosa, N. Papandreou, H. Lung, H. Pozidis, E. Eleftheriou, and C. H. Lam. Recent progress in phase-change memory technology. *JETCAS*, 2016.

[102] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. 19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology. In *ISSCC*, 2014.

[103] Q. Dong, Z. Wang, J. Lim, Y. Zhang, Y. Shih, Y. Chih, J. Chang, D. Blaauw, and D. Sylvester. A 1mb 28nm stt-mram with 2.8ns read access time at 1.2v vdd using single-cap offset-cancelled sense amplifier and in-situ self-write-termination. In *ISSCC*, 2018.

[104] L. Wei, J. G. Alzate, U. Arslan, J. Brockman, N. Das, K. Fischer, T. Ghani, O. Golonzka, P. Hentges, R. Jahan, P. Jain, B. Lin, M. Meterelliyoz, J. O'Donnell, C. Puls, P. Quintero, T. Sahu, M. Sekhar, A. Vangapaty, C. Wiegand, and F. Hamzaoglu. 13.3 a 7mb stt-mram in 22ffl finfet technology with 4ns read sensing time at 0.9v using write-verify-write scheme and offset-cancellation sensing technique. In *ISSCC*, 2019.

[105] J. Li, C. J. Xue, and Yinlong Xu. Stt-ram based energy-efficiency hybrid cache for cmps. In *VLSI-SOC 19*, 2011.

[106] Yong Li, Yiran Chen, and Alex K. Jones. A software approach for combating asymmetries of non-volatile memories. In *ISLPED*, 2012.

[107] David Atienza, Jose M. Mendias, Stylianos Mamagkakis, Dimitrios Soudris, and Francky Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM TODAES*, 2006.

[108] J. Ahn, S. Yoo, and K. Choi. Prediction hybrid cache: An energy-efficient stt-ram cache architecture. *TC*, 2016.

[109] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie. Adaptive placement and migration policy for an stt-ram-based hybrid cache. In *HPCA 20*, 2014.

[110] Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu, Raghu Prabhakar, and Glenn Reinman. Static and dynamic co-optimizations for blocks mapping in hybrid caches. In *ISLPED*, 2012.

[111] Q. Li, J. Li, L. Shi, M. Zhao, C. J. Xue, and Y. He. Compiler-assisted stt-ram-based hybrid cache for energy efficient embedded systems. *TVLSI*, 2014.

[112] Alex Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 2017.

[113] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini. Energy-efficient vision on the pulp platform for ultra-low power parallel computing. In *SiPS*, 2014.

[114] W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater, and D. Atienza. An in-cache computing architecture for edge devices. *TC*, 2020.

[115] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *31st AAAI*, 2017.

William Simon

Address: **Lausanne, Switzerland**
Telephone: **+041 78 644 28 44, +001 954 361 4265**
E-mail: **william.simon@epfl.ch**
Date and place of birth: **1991, USA**

EDUCATION

- 10/2017 – Present **PhD student in electrical engineering/computer science**
  Swiss Federal Institute of Technology, Lausanne, Switzerland
- 09/2015 – 07/2017 **Master of Science in Electrical Engineering**
  Swiss Federal Institute of Technology, Lausanne, Switzerland (GPA: 5.33/ 6.00)
- 05/2010 – 05/2015 **Double Major, B.S. in Electrical Engineering/B.S. in Computer Engineering**
  North Carolina State University, Raleigh, NC, USA (GPA: 3.54 / 4.0)

SELECTED PUBLICATIONS

- *WA Simon, V Ray, A Levisse, G Ansaloni, M Zapater, D Atienza, "Exact Neural Networks from Inexact Multipliers via Fibonacci Weight Encoding", 58th DAC, 2021*
- *WA Simon, M Rios, A Levisse, M Zapater, DA Atienza, "A Memory Chip or Memory Array for Wide-Voltage Range In-Memory Computing Using Bitline Technology", US Patent App. 16/866,566, 2021*
- *WA Simon, A Levisse, M Zapater, D Atienza, "A Hybrid Cache HW/SW Stack for Optimizing Neural Network Runtime, Power and Endurance", 28th VLSI-SoC, 2020*
- *WA Simon, YM Qureshi, M Rios, A Levisse, M Zapater, D Atienza, "BLADE: An in-cache computing architecture for edge devices", Transactions on Computers Vol. 69, 2020*
- *W Simon, J Galicia, A Levisse, M Zapater, D Atienza, "A fast, reliable and wide-voltage-range in-memory computing architecture", 56th DAC, 2019*
- *WA Simon, YM Qureshi, A Levisse, M Zapater, D Atienza, "BLADE: A bitline accelerator for devices on the edge", 29th GLSVLSI, 2019*
- *YM Qureshi, WA Simon, M Zapater, D Atienza, K Olcoz, "Gem5-X: A Gem5-based system level simulation framework to optimize many-core platforms", SpringSim, 2019*
- *Exhaustive publication list may be found on Orcid: https://orcid.org/0000-0001-7357-7204*

HONOURS AND AWARDS

- *Xilinx Open Hardware PhD Prize, Dublin, 2018* – For innovative work with FPGAs on ultrasound devices.
- *Best Paper Award, DASIP, 2017* – A. Ibrahim, W. Simon et. al, "Single-FPGA complete 3D and 2D medical ultrasound imager"
- *Graduated Magna Cum Laude, North Carolina State University, 2015* – Graduated with 3.53/4.00 grade point average.
- *BMW iPower Recognition Award, BMW AG, 2014* – Project saved the department $250,000 a year in human labor.
- *Dean's List, North Carolina State University, 2010–2015* – Received five of eight semesters for outstanding academic work.
- *Goodnight Scholarship, North Carolina State University, 2010–2015* – Competitive academic scholarship for STEM majors.
- *University Scholars Program, North Carolina State University, 2010–2015* – Competitive academic program, requiring rigorous courses to retain membership.

PHD INTERESTS AND PROJECTS

- *In-Memory Computing from Memory Macro to Application Level Implementation* – Initiated a new research stream in lab exploring in-memory computing for SRAM memory. Application level support and benchmarking performed in gem5. Five papers, two patents, and two chips implemented in the 65nm TSMC technology node have been published/fabricated in relation to this project.
- *Hybrid Cache Design and Application Development* – Design of hybrid SRAM/non-volatile memory and supporting allocation algorithm with application level exploration and benchmarking. One paper published.
- *Neural Network Quantization for Embedded Systems* – In relation to the previous projects, neural networks of varying topologies benchmarked on in-memory compute and hybrid caches, with a focus on quantization to improve runtime and power consumption. One paper published.
- *System Level Simulation of Hardware Accelerators* – Utilization of the gem5 simulator to instantiate and simulate the above projects, allowing design space exploration at the application level. Three papers published.

GRADUATE PROJECTS

- *Master Thesis, Profiling the PPP space of MANGO system for HEVC transcoding, Grade of 6.00/6.00, 2017* – Assessed Power, Performance and Predictability (PPP) envelope of HEVC transcoding via software profiling for ARMv7-A and AMD Family 15h architectures. Developed FPGA and algorithmic solutions to software bottlenecks. One paper co-published.
- *Semester Projects, UltrasoundToGo, 6.00/6.00, 2016-2017* – Developed low-latency HDL solutions with team for accelerating ultrasound beamforming algorithms using a Xilinx FPGA. Wrote several modules such as BRAM buffers, voxel adder trees and support for advanced ultrasound functionality. One paper published, five papers co-published.

- *Lab in EDA Based Design, 5.5/6.00, 2016* – Performed full and semi-custom digital and analog projects to gain proficiency in Cadence and Synopsis EDA tools. Projects involved combinational and sequential circuit design as well as analog considerations. Have performed Teaching Assistant duties in this course during PhD.
- *Real-time Embedded Systems, 5.5/6.00, 2016* – Integrated a hardcore+softcore SoC with custom FPGA DMA accelerators to process, in real-time and at low-latency, external stimuli via a button interface, and display results on an external machine via an Ethernet interface. Have performed Teaching Assistant duties in this course during Masters and PhD.

UNDERGRADUATE PROJECTS

- *Senior Project, Entrepreneurial Design, 3.375/4.00, 2014-2015* – Project entailed generation of a feasible, marketable idea, founding a company, advancing the project to a prototype stage, and marketing it to interested parties. Team designed and prototyped product, then developed a marketing strategy to bring it to market.
- *Semester Project, Firefighting Drone Competition, 4.00/4.00, 2015* – Developed with team a drone to assist in firefighting operations. Developed microcontroller/sensor interface and PID algorithms for object avoidance. Team recognized as having most advanced software implementation. Project was extended beyond original finish date due to positive results.
- *L1/L2 Cache Simulator, 4.00/4.00, 2014* – Cache simulator allows user to adjust cache size, block size, and associativity, and feed a list of read/write commands to calculate miss rate and average access time for different configs. Performed design space exploration to find optimal cache size for different command inputs.
- *Tomasulo Pipeline and Reorder Buffer, 4.00/4.00, 2014* – Simulated a Tomasulo pipeline with a reorder buffer and developed a testbench for optimizing configuration for different test applications.
- *Ultrasonic/Light Sensing Car, 3.66/4.00, 2013* – Designed and built car that utilizes ultrasonic and light sensors to navigate a predefined path while avoiding obstacles. Developed and implemented interface circuitry and PID/control code.
- *LC-3 Microcontroller, 4.00/4.00, 2013* – Designed a fully functioning microcontroller in VHDL, including Fetch, Decode, and Execute stages. Runs instruction sets stored to on-chip memory. Synthesized in Quartus II and simulated in Modelsim.

WORK EXPERIENCE

- 07/2016 – 09/2016 **Embedded Systems Engineer – Xaplos Inc.**
  - *E-DAQ1 Product Example Kit* – Wrote drivers for board accelerometer, gyroscope, LCD screen, digital and analog inputs and outputs, CAN bus, and light, pressure, temperature, humidity, thermocouple, resistance, and current sensors.
  - *Blood Pressure Monitor* – Designed interface between microcontroller and cuff pressure sensor and pump, as well as algorithms to calculate systolic and diastolic blood pressure from sensor readings.
- 09/2012 – 05/2014 **BMW Co-op, Three Semester-long Internships**
  - *Component Verification Program* – Compares official car bill of materials to plant build lists and reports inconsistencies such as double part demand, incorrect part location, etc. Caught 10+ major errors as of May 2014 that prevented costly line failures and rework.
  - *Pipeline75* – Program that interfaces with US and German SAP databases to pull information for all 150,000+ parts relevant to the plant. Scope of parts can be adjusted by introduction time and car model. Reduced time for data acquisition time, previously 8 hours, by 13x.
  - *Deviation Tracking Software* – Automation of manual processes by integrating SharePoint, Excel, and SAP code. Estimated to save the department in excess of $250,000 a year.

SELECT COURSES/SPECIALTIES

- Advanced Multiprocessor Architecture Design
- Analog Circuits Design I, II
- VLSI Design I, II
- Biomedical Signal Processing
- Advanced Multiprocessor Architecture Design
- Design of Complex Digital Systems
- Information Theory
- ADC/DAC Design
- Bio-Nano Engineering
- Entrepreneurial Thinking, Engineering Entrepreneurship
- Embedded System Design I, II, Realtime Embedded Systems
- Nanoelectronics
- Mechatronics (Microtechnique)
- Communications Engineering
- French Level B1/B2

SOFTWARE/HARDWARE EXPERIENCE

- C/C++
- Assembly
- Python/PyTorch
- Verilog/VHDL
- Matlab
- Java
- AI/Neural Networks
- Memory Design
- Blockchain
- Altera FPGAs/Software
- Xilinx FPGAs/Software
- Cadence/Mentor Design Tools