Heterogeneous System Acceleration: Simulation and Exploration

CprE 581 Final Project

Braedon Giblin
bgiblin@iastate.edu

Matthew Dwyer
dwyer@iastate.edu

December 16, 2021

Abstract

Heterogeneous processors are increasingly becoming the vector in which modern architects achieve maximum performance. Utilizing multiple cores coupled with several accelerator units, higher performance and energy efficiency can be realized on a particular workload. Despite the significant push for heterogeneous architectures, ensuring that those designs will perform prior to design still remains a prominent challenge. Our work implements a model for a heterogeneous system based on a simple embedded matrix multiply benchmark. We will run this benchmark on a reference machine, and measure / calculate baseline performance and power metrics. Then, we implement a simple accelerator into the system and re-record power and energy characteristics.

1 Introduction

Embedded systems have grown increasingly complex as machine learning, computer vision, and high speed data processing requirements needed for tasks such as automated driving, image inference, speech recognition, and big data analysis have taken off. The need for embedded systems that can accomplish these computationally complex tasks while performing in an energy and computationally constrained environment is critical for the future development of novel embedded applications. As embedded processors and SoCs continue to expand their capabilities, innovative workloads are developed to utilize the capabilities of those new devices. Applications once unobtainable by desktop or server scale machines are now finding new applications on embedded devices, paving the way for additional specialized hardware integrated into embedded device architecture.

Accelerated processing is largely considered the future vector that should be utilized to improve the performance of embedded processors [14]. By utilizing accelerators coupled with the main core, embedded processors are able to tackle computationally difficult problems that traditionally required more powerful processors — such as computer vision, ML inference, and multimedia processing. Furthermore, these accelerators not only improve the performance for application specific embedded processors, but can also reduce the power requirements of them. This is done by allowing for a slower general purpose core implementation and offloading tasks to higher efficiency cores, functional units,

or co-processors, reaping speed benefits from the power optimized, domain specific accelerator.

This work aims to evaluate one such platform by modeling and comparing a possible embedded design that may benefit from such acceleration. This acceleration is modeled via the gem5-X simulator — an extension of the industry standard gem5 architecture simulator [2] that supports modeling heterogeneous systems and accelerators. By providing comprehensive models of potential designs, we show the performance changes that occur with various architectural implementations.

2 Related Work

Heterogeneous acceleration has been a hot research topic for several y ears, including in the embedded domain. Research has shown that effectively using acceleration of common operations on an embedded system can actually substantially improve the energy efficiency of the system [6, 10, 9]. Many implementations of domain specific functions, including machine learning training and inference, multimedia processing, digital signal processing, and vision processing have proven to provide substantial performance gains. These performance gains can manifest themselves in numerous metrics, primarily *power consumption*, *data throughput/latency* (performance), and *die area*.

2.1 Power Efficiency

Power and energy consumption have always been of particular concern in embedded devices [11]. While processing demands of embedded applications have risen, battery innovation continues to slow in relation [4]. Machine learning is a prime workload candidate to be moved on-device due to the high power cost of off-device communication. Yet, performing a single inference in a basic ML model requires substantial computational effort and power on a general purpose low-power embedded core. Convolution and matrix multiplication are optimal candidates for acceleration due to their general popularity, repetitious computations, and large power costs. Sun et al. demonstrates the improvements afforded by hardware specific acceleration structures for Convolution Neural Network (CNN) based image classifiers as one of many examples [15].

Instruction Set Architecture is also key in enabling simple and power-efficient designs at the lowest levels. Reduced Instruction Set Computers (RISC) focus on breaking computation down into basic computational building blocks and constructing complex procedures from their foundation. Conversely, Complex Instruction Set Computers (CISC) aim to support basic and advanced options at the assembly level, either by adding in acceleration structures or decomposing them into many μ-operations. While ISA is irrelevant of power and performance above moderately performant modern processors (Cortex-A9), at the very-low performance tier ISAs still play a role. Due to the large overheads of the x86 architecture, low-performance embedded cores often implement a subset of the

ARM ISA to reduce architecture complexity (opting for in-order pipelines with minimal TLB, prediction, speculation, and pre-fetching support) [1].

2.2 Performance

Machine Learning applications can be challenging to map to embedded devices due to latency and throughput requirements and the resource intensive operations they preform [13]. Precision and accuracy of the model's inferences are primary trade-offs of performance, yet not all use-cases can take advantage of such benefits—autonomous navigation, healthcare, and security applications included. These optimizations include quantizaiton/reduced precision computation, model pruning, reduced input resolution, and model architecture simplification [13]. While model complexity can be reduced, machine learning still remains primarily memory bound on embedded devices.

High memory usage is caused by the large volume of data movement required for ML inference. Embedded devices also often see drop in performance attributed to reduced cache sizes and increased memory latencies. Large model sizes also often require substantial compression to fit on even moderate embedded devices. Model weights/kernels, input, and output feature maps all vie for limited cache resources, causing significant numbers of misses, evictions, and writes to off-chip memory, incurring performance and power penalties. While architectural optimizations can leverage data reuse to reduce these penalties, multi-layer or multi-model systems stress these implementations even further [17]. Additionally, memory access latency and bandwidth on embedded and low-power memory hierarchies significantly under-perform their desktop or high-end mobile counterparts. While desktop and server class architectures afford significant power and area to speculation and IPL structures, these techniques are either minimized or removed from simple embedded designs. This significantly reduces a processor's ability to hide long latency operations such as memory accesses. Reduced cache sizes and low-power memory hierarchies only exacerbate this issue, forcing performance gains to stem from computational efficiency [5].

Multiply accumulate (MAC) operations are fundamental to matrix multiplication and convolution kernels. While simple, combining floating-point/integer multiplication and addition into a single ALU operation can lead to radical improvements in processing efficiency. Accelerated hardware structures of MAC Functional Units (FU) — such as systolic arrays — can massively accelerate matrix operations and thus machine learning inference [7, 3]. Streaming architectures are also popular for complex convolutions in CNNs or even fully-connected operations, utilized heavily in machine vision and audio applications [18].

2.3 Die Area

Reducing die footprint can be crucial to reducing production cost, verification difficulty, and power consumption. As complexity and die size increase, so does

the transistor count and power consumption (assuming process node is held constant). While general purpose cores allow for the least complexity and largest variety of software capabilities, performance falters for computationally repetitive applications. Yet, when accelerators are introduced die area increases. Despite that, accelerators by nature are designed to better utilize their power budget to perform domain specific operations. While predominately implemented though on-chip accelerators, heterogeneous embedded architectures also take advantage of external accelerators such as GPUs [17, 9].

3 Main Idea

Our primary goal was to verify our expected performance improvements in representative machine learning benchmarks on embedded devices with the inclusion of a matrix-multiplication accelerator. Due to the ubiquity of the ARM instruction set and ARM designed processors, we desired to perform these tests on accurately modeled common ARM embedded CPUs. We also wanted to compare in-order and out-of-order core architectures. While matrix computations are generally memory bound rather than compute, we felt it would provide a good verification of this fact. In an attempt to combat these bottlenecks, we also opted to adapt our benchmark to a tile our benchmark to better map it to our accelerator, install High Bandwidth Memory (HBM), and also perform floating point operations. These changes saw varying results which can be seen in our results table.

4 Methodology

For all simulations we utilized the Gem5 cycle-accurate system-level simulation platform. This platform allowed us to vary processor architecture and tweak our CPU and accelerator parameters, providing a diverse set of results for us to compare against. Below you can see the table of CPU configurations that we utilized for our ARM Cortex-A8 in-order and ARM Cortex-A9 out-of-order processors. These values are based of a gem5 tuning paper which sought out to model both of the aforementioned processors in Gem5 for accurate simulations of embedded device workloads. While we attempted to follow the configurations provided in the paper as close as possible, ultimately, some configurations were challenging to implement, and thus were omitted in our results. While this reduces the accuracy of our simulations in relation to real-world performance of the A8 and A9 processors, for our comparisons these deficiencies could be disregarded, as they were held constant between all of our simulations. Where configuration values were not provided or stated, the default Gem5 O3 (out-of-order) and MinorCPU (in-order) configurations were used. A table of our augmented processor values can be seen in Table 1 Gem5 also allowed us to utilize our desired ARM ISA for our simulations.

Table 1: Configurations for Each Processor

Parameter			Cortex-A9	Cortex-A8
Core Clocks			800 MHz	800 MHz
DRAM	Size		256 MB	256 MB
	Clock		400 MHz	166 MHz
	Latency		65	65
L2	Size		512 kB	256 kB
	Associativity		8	8
	Latency		8	8
	MSHRs		11	16
	Write Buffers		9	8
L1-I	Size		32 kB	32 kB
	Associativity		4	4
	Latency		1	1
	MSHRs		2	1
L1-D	Size		32 kB	32 kB
	Associativity		4	4
	Latency		1	1
	MSHRS		2	1
	Write Buffers		16	1
	Strided Prefetchers	Degree	N/A	N/A
		Buffer Size	N/A	N/A
Global BP	Entries		4096	512
	Bits		2	2
BTB Entries			4096	512
Return Address Stack			8	8
ITLB/DTLB entries			64/32	64/32
Issue Width			4	2
gm5 effective execution stage depth (wbDepth)			8	8
Pipeline Stages			8	8
Physical INT/FP registers			256/256	N/A
IQ entries			32	16
LSQ entries			12	12
ROB entries			192	N/A

The ARM ISA was used as our processor architecture and compile target. This was for a number of reasons, primarily due to the prominence of it in embedded devices. This is in part due to the Reduced Instruction Set Computer (RISC) methodology it follows in selecting what instructions processors must support. RISC allows for embedded processors and SoCs to implement simpler hardware decoders and control-flow, reducing power usage and die area and in some cases increasing performance.

We initially planned to utilize the Gem5 cycle-accurate processor simulator

with the Gem5-X extensions. While this seemed promising at first, it ultimately had goals that were distinct from our own. The Gem5-X extensions focused on providing support for multi-core systems with variable core architectures (Think BIG.little, or Power and Performance core systems). It also provided minimal support for in-cache data computation and 3D stacked HBM memory systems. While we ultimately did run simulations of systems containing HBM (LPDDR5) DIMM models, we opted for the models provided in the base Gem5 libraries rather than those provided by Gem5-X.

The benchmark we utilized was the integer matrix multiplication (*matmult-int*) application which is a part of the Embench embedded systems benchmark suite. This application takes two 30x30 integer matrices and multiplies them by one-another. This matrix-matrix multiplication is derived from many multiply accumulate operations within loops to compute the final result. This benchmark is representative of machine learning applications, as matrix operations are utilized to perform layer computations in almost all layer types utilized in modern machine learning (deep learning fully connected layers, convolution layers, etc.). As a result, this benchmark should give us a general idea of the performance of machine learning applications on our simulated processors.

Our initial plan was to implement the accelerator as an independent functional unit within the processor. This was purely done for the sake of convenience with adding the instruction and collecting stats. As such, our "functional unit" would behave closer to a co-processor. The basic architecture provided the accelerator with a block of dedicated, low-latency SRAM. This SRAM was visible from global memory space, essentially enabling the programmer to load and store into the accelerator via *memcpy*. We assumed that the time to load and store into the buffer was equal to the time to load and store data locally. This SRAM then could be accessed with an assumed 1 cycle of latency by the accelerator, as the accelerator performed the matrix multiplication and stored the result back to the buffer. The accelerator would take a varying amount of time to complete the task, and would be non-pipelined, setting and clearing a busy flag in a status register that software could poll to perform the next multiplication.

In the end, we couldn't get our accelerator to work as intended. So, we elected to model our accelerators latency just using C heuristics. This enables us to see the speedup from using our accelerator by summing the memory latency and the FU latency. We estimated that a reasonable latency for a 30x30 matrix-matrix multiplication would be 80 clock cycles. This assumes that we are using an arbitrarily large systolic array. We estimate that it should take 60 clock cycles for loading and draining the array, and then we add an additional 20 to account for preparing outputting the data to the unified buffer.

While this model is far from perfect, we think it should give us a reasonable idea of performance gains using an accelerator. Had we more time, we would have liked to model our entire accelerator as described in gem5, though using this shortcut enabled us to get semi-accurate results.

5 Results

System Configuration	Benchmark	ROI (sec)	CPI	IPC
ARM Cortex-A8	Int	0.4217	1.052	0.951
	Tiled Int	0.4336	1.073	0.932
	Float	0.4217	1.052	0.951
	Tiled Float	0.4336	1.073	0.932
ARM Cortex-A9	Int	0.0747	0.873	1.146
	Tiled Int	0.0747	0.873	1.146
	Float	0.2646	0.660	1.516
	Tiled Float	0.2685	0.665	1.505
ARM Cortex-A8 + HBM	Int	0.4217	1.052	0.951
	Tiled Int	0.1281	1.488	0.672
	Float	0.4217	1.052	0.951
	Tiled Float	0.4336	1.073	0.932
ARM Cortex-A9 + HBM	Int	0.0747	0.873	1.146
	Tiled Int	0.0747	0.873	1.146
	Float	0.2646	0.660	1.516
	Tiled Float	0.2685	0.665	1.505
ARM Cortex-A8 + 30x30 Accelerator	Int	0.0228	4.407	0.227
	Tiled Int	0.0228	4.407	0.227
	Float	0.0312	5.090	0.196
	Tiled Float	0.0313	5.090	0.196
ARM Cortex-A9 + 30x30 Accelerator	Int	0.0117	2.279	0.439
	Tiled Int	0.0117	2.279	0.439
	Float	0.0179	2.948	0.339
	Tiled Float	0.0179	2.948	0.339

Table 2: Simulation results for each CPU configuration on our suite of benchmarks. Results provided are the Sim Time of our ROI in seconds, our Cycles Per Instruction for our ROI, and our Instructions Per Cycle of our ROI.

Our first experiment was how a basic embedded system implementation might implement our benchmark. We elected to try two separate processors of varying architecture (in-order & out-of-order) to represent potential real world processor selections for our workload. We modified the the benchmark program to support 4 different run modes: Integer, Float, and tiled variants of each. The tiled variants were in an effort to better map access to our small cache sizes in response to initial results yielding execution times indicative of our memory bound application.

We then ran these through each of our systems: the A8, the A9, accelerated variants, and with HBM. We hoped this would give us a good spread of results to see where we get performance benefits. With our HBM configurations, we were hoping to see the extent to our memory bound nature, and expected to see large improvement. Ultimately, this was not the case. See Table 2 for complete results.

We noticed that the Out of Order processor performed significantly faster

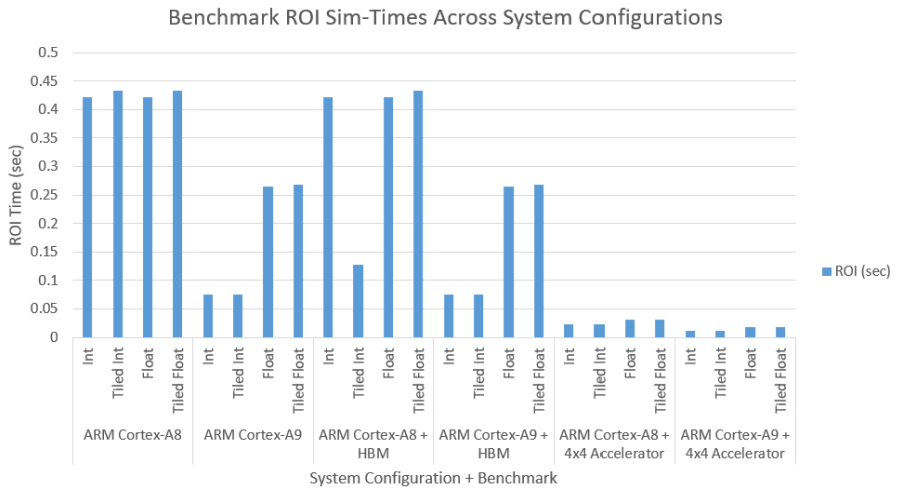

Figure 1: Benchmark ROI Sim Time across our system configurations.

than the in order processor. This was not very surprising to us as the the O3 core is able to hide the memory latency associated with loading and storing the matrices much better. Our caches were slightly larger for our A9 core as well. This likely also reduced memory latency as well by reducing compulsory misses.

Our accelerator outperformed both processors significantly. This is likely largely because we removed a significant amount of memory latency, as we assume that once our matrices are loaded into the accelerator's unified memory buffer, we can then access the data from the accelerator in 1 cycle. We experimented with slightly modifying the accelerators latency, and saw relatively subtle effects from that modification.

The HBM A8 & A9 system implementations surprised us the most. While our application is primarily memory bound, including a significantly faster and wider memory bus did not produce the significant speed-up we expected. While in some cases such as the A8 + HBM Tiled Int benchmark we saw x3 improvement over the standard A8 implementation, these cases were few an far between the majority that yielded identical runtimes. This is assumed to be caused by the size of our two level cache hierarchy being large enough to fit our data and results with few collisions, requiring few accesses to the memory after they have been warmed by our benchmark prior to our ROI.

Another data point of note which is clear in Figure 2 is the increase of CPI and decrease in IPC of our systems with a dedicated accelerator. This is assumed to be due to the omission of the math operations (floating-point and integer) of the matrix-matrix multiplication. The ratio of memory to ALU instructions is far greater in these simulations, thus, the CPI and IPC results show relative increases. It can also be seen that the floating-point benchmarks have consistently higher CPI and IPC values with an accelerator than the sim-

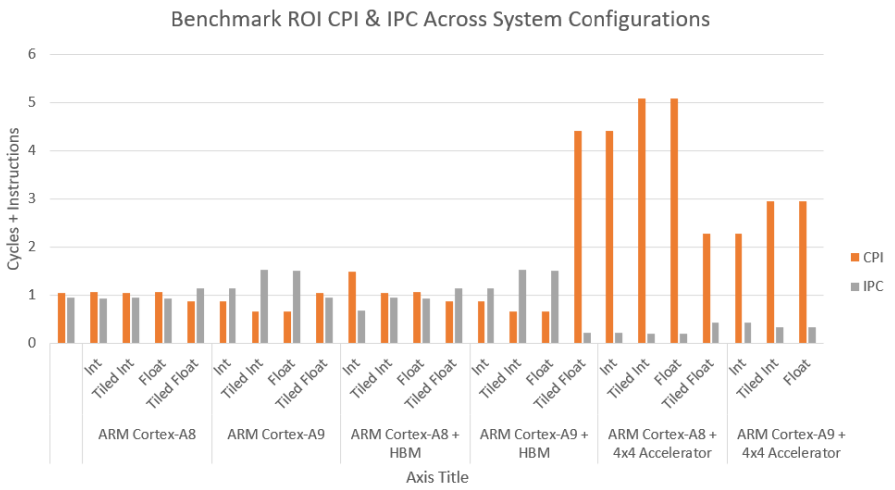

Figure 2: Benchmark ROI CPI & IPC across our system configurations.

ulations with no accelerator (Baseline or with HBM). We are unsure why this is the case, although data size may be playing a factor, depending on the libraries at use, since our data types are C++ *long* and *float* values for our integer and floating-point benchmarks respectively.

Our results show that an accelerator for even small matrix multiplications could be highly effective at improving the performance of the benchmarked applications for both floating-point and integer values. This performance trade off would likely change if our unified memory became smaller than the array we were attempting to multiply. If our memory or accelerator were to be smaller than our 30x30 test size, this would have required numerous accesses and intermediate computations to correctly compute.

6 Future Work

One of the primary issues seen with our benchmarks was memory transaction overhead. Matrix multiplication as we have shown is a memory-bound operation [17], and thus despite additional accelerated structures, performance increases remained marginal. Thus, in-memory-computation would likely further reduce this bottle neck. By removing the additional overheads of memory transaction latency, we would expect to see significant improvements in performance, although the size of the accelerated structures capable of being implemented within the memory hierarchy still remains an open question.

Floating-point operations would be another logical area of future works. While integer operations (multiplication and addition) are abundantly optimized and performant, floating-point operations require significantly higher

computational logic to perform the same basic operations. While this isn't to imply that the same benchmark if implemented with floating point operations would no longer be memory bound, it would likely show a vast difference in performance capabilities. This is especially true due to the nature of floating-point operations on ARM CPUs. Until ARMv8, all floating-point workloads were done utilizing an IEEE 754-1985 implementation software library [8]. Hardware support for floating-point operations can dramatically increase performance of these operations [12, 16], thus considering floating-point operations might yield varied results compared to our findings.

Finally, power efficiency is a vital component of embedded hardware design. Initially we set out to provide analysis of how machine learning (integer matrix multiplication) embedded accelerators can co-exist within embedded architectures while still remaining within thermal and power envelopes. This additional analysis was hindered by the time and tools at our disposal, thus we were not capable of performing such analysis the necessary precision. Comparing the size of the accelerator and its energy consumption against performance differences would provide meaningful insight into the feasibility of such structures, and further promote their inclusion in the embedded processing world.

7 Conclusion

Our results matched our initial expectations about the ROI execution time when including an matrix-matrix accelerator in our CPU. With our 30x30 accelerator present, we saw significant increases in program execution over both our integer and floating-point benchmarks (Tiled and Regular). Due to the memory-bound application, we expected larger returns from our HBM memory, but saw little to no performance improvement. This is likely due to the small data sizes fitting in the cache.

These results show that for matrix-matrix operations, including machine learning inference and training workloads, including an accelerator such as ours can significantly speed-up execution time on medium to small matrix sizes. We expect that our results would scale to larger matrices as well, although data could then become an issue due to the limited cache hierarchy.

Overall, this project was challenging in a multitude of ways. Not only was implementing the tuned ARM Cortex-A8 & Cortex-A9 processors a larger hurdle than first envisioned, but implementing our accelerator was far less trivial than expected. Our goals ultimately caused us to refocus our implementation intentions and understand Gem5, Gem5-X, and the Embench suite in far greater detail. We unintentionally ran a larger gamut of tests than initially proposed, and achieved results that matched our expectations.

References

[1] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2013.

[2] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy evaluation of gem5 simulator system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, July 2012.

[3] Hyungmin Cho. Risa: A reinforced systolic array for depthwise convolutions and embedded tensor reshaping. *ACM Trans. Embed. Comput. Syst.*, 20(5s), September 2021.

[4] Sheridan Few, Oliver Schmidt, Gregory J. Offer, Nigel Brandon, Jenny Nelson, and Ajay Gambhir. Prospective improvements in cost and cycle life of off-grid lithium-ion battery packs: An analysis informed by expert elicitations. *Energy Policy*, 114:578–590, 2018.

[5] Bastian Florentz and Michaela Huhn. Embedded systems architecture: Evaluation and analysis. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Quality of Software Architectures*, pages 145–162, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Bogil Kim, Sungjae Lee, Amit Ranjan Trivedi, and William J. Song. Energy-efficient acceleration of deep neural networks on realtime-constrained embedded edge devices. *IEEE Access*, 8:216259–216270, 2020.

[7] Christos Kyrkou and Theocharis Theocharides. Scope: Towards a systolic array for svm object detection. *IEEE Embedded Systems Letters*, 1(2):46–49, Aug 2009.

[8] ARM Limited. Arm compiler arm c and c libraries and floating-point support user guide version 6.13.

[9] Abhinandan Majumdar, Srihari Cadambi, and Srimat T. Chakradhar. An energy-efficient heterogeneous system for embedded learning and classification. *IEEE Embedded Systems Letters*, 3(1):42–45, 2011.

[10] Maria Malik, Farnoud Farahmand, Paul Otto, Nima Akhlaghi, Tinoosh Mohsenin, Siddhartha Sikdar, and Houman Homayoun. Architecture exploration for energy-efficient embedded vision applications: From general purpose processor to domain specific accelerator. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 559–564, July 2016.

[11] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, April 2001.

[12] V Nikolskiy and V Stegailov. Floating-point performance of ARM cores and their efficiency in classical molecular dynamics. *Journal of Physics: Conference Series*, 681:012049, feb 2016.

[13] Wolfgang Roth, Günther Schindler, Matthias Zöhrer, Lukas Pfeifenberger, Robert Peharz, Sebastian Tschiatschek, Holger Fröning, Franz Pernkopf, and Zoubin Ghahramani. Resource-efficient neural networks for embedded systems, 2020.

[14] M. Schlett. Trends in embedded-microprocessor design. *Computer*, 31(8):44–49, 1998.

[15] Baohua Sun, Lin Yang, Patrick Dong, Wenhan Zhang, Jason Dong, and Charles Young. Ultra power-efficient cnn domain specific accelerator with 9.3tops/watt for mobile and embedded applications. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[16] Liang-Kai Wang, Charles Tsen, Michael J. Schulte, and Divya Jhalani. Benchmarks and performance analysis of decimal floating-point applications. In *2007 25th International Conference on Computer Design*, pages 164–170, Oct 2007.

[17] Ying Wang, Huawei Li, and Xiaowei Li. Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, Nov 2016.

[18] Ruizhe Zhao, Wayne Luk, Xinyu Niu, Huifeng Shi, and Haitao Wang. Hardware acceleration for machine learning. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 645–650, July 2017.