Project Proposal for ECE751: Embedded Computing Systems

An Efficient Architectural Realization of a Specialization Engine for Neural Networks Retaining General-Purpose Programmability

Vinay Gangadhar Sharmila Shridhar Anil Ranganagoudra Chandana Hosamane
{gangadhar, sshridhar, ranganagoudr, hosamanekabb}@wisc.edu

Abstract

With the traditional scaling laws – Dennard's and Moore's law slowing down in recent years, a special class of accelerators called Domain Specific Accelerators (DSAs) are being explored to reap performance and energy efficiency for particular embedded application domains. Though DSAs obtain 10x to 1000x performance and energy benefits compared to a general purpose processor, they compromise programmability and are prone to obsoletion due to domain volatility, and also incur high recurring design and verification costs. This necessitates the need for another class of accelerators called Programmable Accelerators which retain the programmability for different application domains and try to achieve the performance, area and energy efficiency of each DSA. The main insight we have is that DSAs employ certain common specialization principles across various application domains and we identify them as – Concurrency, Specialized Computational Units, Communication, Data-reuse and Co-ordination. *We have studied these specialization principles in detail and have already proposed a programmable engine employing these principles. Based on some preliminary modeling results on application domains like deep neural networks (DNNs), convolutional neural networks (CNNs) and database processing we have finalized a high-level architectural organization of this programmable engine.*

For this course project, we mainly want to target neural network domain and build/improvise the specialization engine with support for programmability. It should be able to execute wide variety of neural network applications like convolution, speech recognition and text recognition kernels. We call this engine as Programmable Engine for Neural Networks (PENN). The aim of this project is i) to explore the trade-offs of many fine-grain design decisions to be taken for PENN based on a thorough analysis of neural network workloads, ii) finalize the micro-architectural design details, iii) build the end-to-end software toolchain to compile the programs and configure PENN and iv) realize the prototype on a Zynq FPGA. We want to evaluate our design for performance, power and area with state-of-the art neural network DSAs. We want to limit our study only to neural networks for this project, although the preliminary modeling is done for other domains too.

1. Motivation

Diminishing gains of transistor scaling [4, 20, 5] has been responsible for the trend moving towards Domain Specific Accelerators (DSA) in past few years. DSAs trade-off general purpose programmability for gains in performance, energy and area. There are several DSAs proposed for various application domains [3, 13, 22, 2]. This lack of flexibility makes DSAs prone to obsoletion with constantly evolving algorithms and standards to process data. Most modern devices run varied workloads, which makes it necessary to have multiple DSAs in devices, thereby making the combined system un-optimal and consume a larger area. General purpose processors are at the other end of the spectrum which are capable of running any workload but cannot get efficiency gains as DSAs. Reconfigurable architectures like Dyser [8], Wavescalar [21], TRIPS [18] are proposed in past few years which try to match the efficiency of specialized hardware engines while maintaining the programmability.

We propose similar programmable architecture for neural network domain and try to achieve efficiency of a neural network DSA while maintaining programmability. We propose a programmable engine for neural networks (PENN). With the proposed PENN architecture, any neural network program can be executed while still getting the efficiency of custom DSA solution. The factors that effect the design choices in DSAs are identified and realized in the PENN architecture. We explain the specialization principles commonly found in DSAs and employ the same for PENN and derive at the architecture. Our insight is that PENN could achieve the performance of a DSA specific to each application while maintaining the programmability.

2. Background and Proposed PENN Architecture

Before explaining the high level organization of PENN architecture, we define the five specialization principles we employed to consider this style of architecture. Our primary insight on coming to the architectural substrate is based on well-understood mechanisms of specialization used in DSAs. We first explain the assumptions we make for neural network workloads based on our preliminary analysis.

Workload Assumptions 1. Neural network workloads have significant parallelism, either at the data or thread level. 2.

They perform some problem specific complex computational work and not just data-structure traversals. 3. They have coarse grain units of work. 4. They have regular memory access patterns.

2.1. Specialization Principles

Broadly, we see these principles as a counterpart to the insights from Hameed et al. [10], in that they describe the sources of inefficiency in a general purpose processor, whereas our findings are oriented around elucidating the sources of potential efficiency gain from specialization.

Concurrency Specialization The concurrency of a workload is the degree to which its operations can be performed in parallel. This concurrency can be derived from data or thread level parallelism found in the workloads. Examples of specialization strategies include employing many independent processing elements with their own controllers, or using a wide vector model with a single controller. We chose the former one as baseline architecture having many processing elements with a low-power controller.

Computation Specialization Computations are individual units of work in an algorithm executed by functional units (FUs). Specializing *computation* means creating problem-specific FUs. For instance, a `sin` FU would much more efficiently compute the sine function than iterative methods on a general purpose processor. Specializing computation improves performance and energy by reducing the total work. Most of the neural network applications employ some commonality in FU types.

Communication Specialization Communication is the means of transmission of transient values between the storage and functional units. Specialized communication is simply the instantiation of communication channels, and potentially buffering, between FUs to ultimately facilitate a faster operand throughput to the FUs. This reduces power by lessening access to intermediate storage, and potentially area if the alternative is a general communication network.

Data Reuse Specialization Data reuse is an algorithmic property where intermediate computed values are consumed multiple times. The specialization of data reuse means using custom storage structures or reuse buffers for these temporaries. Specializing reuse benefits performance and power by avoiding the more expensive access to a larger global memory or register files.

Coordination Specialization Hardware coordination is the management of multiple hardware units and their timing to perform work. Instruction sequencing, control flow, interrupts handling and address generation are all examples of coordination tasks. Specializing it usually involves the creation of small finite state machines to perform each task. A low-power in-order core or a micro-controller could be used for this coordination specialization.

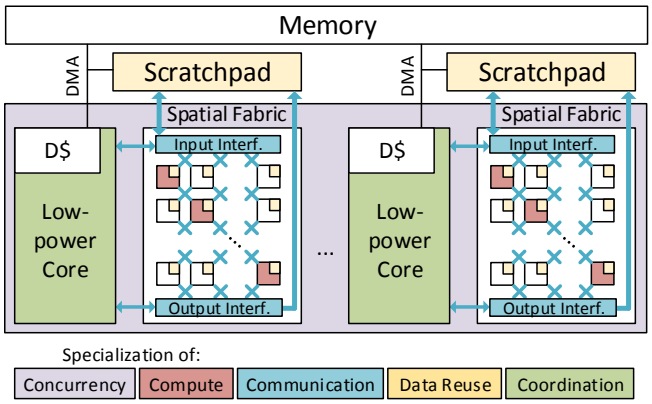

Figure 1: Programmable Engine for Neural Networks (PENN) Architecture

2.2. PENN Architecture

Figure 1 gives a high level overview of the PENN architecture we have proposed. The requirement to exploit high concurrency in workloads pushes the design towards simplicity, and the requirement of programmability implies the use of some sort of programmable core. The natural way to fill these combined requirements is to use an array of tiny low-power cores which communicate through memory. These low power cores have caches, general purpose programmable and are basic building blocks of PENN. The low-power core along with other specialization units (explained below) form one unit of PENN fabric and multiple such units are used to specialize *Concurrency*. Using many units, as opposed to a wide-vector model, has a flexibility advantage. When memory locality is not possible, parallelism can still be achieved through multiple execution threads. The remainder of the design is to straightforwardly apply the remaining specialization principles.

Achieving *communication* specialization of intermediate values requires an efficient distribution mechanism for operands, which avoids expensive intermediate storage like multi-ported register files. Arguably, the best known way to do this is to use an explicit routing network, which is exposed to the ISA to eliminate the hardware burden of dynamic routing. This property is what defines spatial architectures and therefore we add a spatial architecture as our first mechanism. The spatial architecture can be a Coarse Grain Reconfigurable Fabric (CGRA) which has an intermix of FUs connected spatially through an interconnected network. This serves two additional purposes. First, it is an appropriate place to instantiate custom functional units, i.e. *computation* specialization. Second, it accommodates *reuse* of constant values associated with specific computations. In principle, this spatial architecture can be either fine-grain reconfigurable (FPGA) or more coarse grain reconfigurable.

To achieve *communication* specialization with the global memory, a natural solution is to add a DMA engine and configurable scratchpad, with a vector interface to the spatial architecture. The scratchpad, configured as a DMA buffer, enables

the efficient streaming of memory by decoupling memory access from the spatial architecture. When configured differently, the scratchpad can act as a *reuse* buffer. In either context, a single-ported scratchpad is enough, as access patterns are usually simple and known ahead of time.

Finally, we require an efficient mechanism for *coordinating* the above hardware units (e.g. configuring the spatial architecture or synchronizing DMA with the computation). Again, here we propose relying on the simple core, as this brings a huge programmability and generality benefit. Furthermore, the cost is low; if the core is low-power enough, and the spatial architecture is large enough, the overheads of coordination can be kept low.

To summarize, each unit of our proposed architecture contains a Low-power core, a Spatial architecture, Scratchpad and DMA. This architecture satisfies the programmable accelerator requirements: general-purpose programmability, efficiency through the application of specialization principles, and simple parameterizability.

PENN in Practice Preparing the PENN for use occurs in two phases: 1. *design synthesis* – the selection of hardware parameters to suit the chosen workload domain; and 2. *programming* – the generation of the program and spatial datapath configuration to carry out the algorithm.

For our project, though many optimization strategies are possible, we want to consider the primary constraint of this programmable architecture to be performance – i.e. there exists some throughput target that must be met, and power and area should be minimized, while still retaining some degree of generality and programmability. We want to explore micro-architectural design decisions needed to make this design synthesis step easier and efficient for many workload kernels.

Programming PENN has two major components: creation of the coordination code for the low power core and generation of the configuration data/stream for the spatial datapath to match available resources. In practice using standard languages with `#pragma` annotations, or even languages like OpenCL would likely be more effective. Though we do not aim to implement a full-working compiler in this project, we want to develop an API for PENN programming and hand-generate assembly instructions along with configuration stream to configure PENN. Figure 2 shows an example annotated code for computing a neural network layer, along with a provisioned (already synthesized) PENN. The figure also shows the compilation steps to map each portion of the code to the architecture. At a high level, the compiler will use annotations to identify arrays for use in the SRAM, and how should they be used (either as a stream buffer or scratchpad). In the example, the weights can be loaded into the scratchpad, and reused across invocations.

Subsequently, the compiler will unroll and create a large-enough datapath to match the resources present in the spatial fabric, which could be spatially scheduled using scheduling

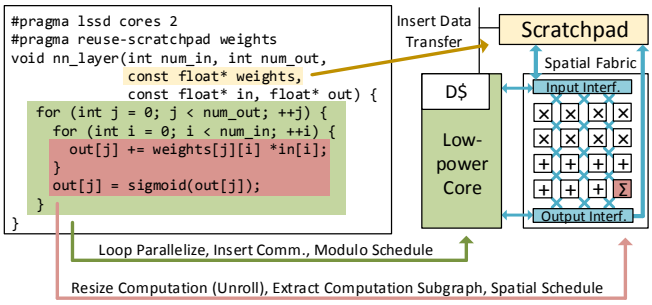

Figure 2: Example Program & Compiler Passes
(Arrows labeled with required compilation passes.)

techniques. Communication instructions would be inserted into the coordination code, as well as instructions to control the DMA access for streaming inputs. Finally, the coordination loop would be modulo scheduled to effectively pipeline the spatial architecture.

3. Project phases

We explain the phases of project here and expected timeline for each step. Some of the phases can be carried out in parallel and will be distributed among the project members.

- **Phase 0 [1 week]:** Choose the representative workloads/ kernels of neural network domain targeting different applications (image processing, speech recognition, text parsing etc.,). Use the trace based modeling tool TDG ([17]) modeled for PENN to determine the initial performance and power estimates for each kernel. More accurate analysis of workloads (profiling) to be done here and see if there are any missing architectural pieces not considered for current PENN architecture.
- **Phase 1 [2 days]:** Based on phase 0 analysis, some important design trade-off decisions should be taken. Those steps are listed here: i) Low-power core: Properties (pipeline stages, memory interface and hierarchy etc.,) of low-power core which does the co-ordination. Compiler and software toolchain for the core (RISCV toolchain and their in-order core can be a good start). ii) CGRA: Types of problem specific FUs inside the fabric. Interface to low-power core and scratchpad memory. Addressing of scratchpad space and global memory. Scheduling pattern for CGRA. Scratchpad size and bit-width. iii) Programming model: API for PENN. Pragmas and code annotations for compilation.
- **Phase 2 [3 weeks]:** This phase involves implementation of individual modules of PENN listed in Phase 1. Implementation of low power core and CGRA in Verilog and C++. Writing API routines for PENN and compiler support (Note: A full-fledged compiler may not be implemented but a framework to generate instructions and configuration stream will be implemented.) Tools needed for the project are explained in Section 4.
- **Phase 3 [1 week]:**This phase mainly involves evaluating implemented synthesized PENN architecture with representative workloads chosen in Phase 0. We also use C++

simulator written for PENN to correlate the functionality of the synthesized version. We plan to evaluate PENN for performance, area and power against a state-of-the-art DSA for Neural Network.

- **Phase 4 [2 weeks]:** This phase mainly involves prototyping PENN on Zynq FPGA. However, this phase will be realized only if Phase 3 is completed well within course project time limit. Project report is also part of this phase.

4. Methodology

We try to list out some of the tools we are considering for our project and also the methodologies involved with those.

We plan to use existing RISCV LLVM toochain [15] as part of compiler framework. For low power core, we are considering a three stage pipeline SODOR core as it is power efficient compared to any standard VLIW core. For implementing individual modules, we want to use CHISEL [1] tool, which uses an embedded scala language to generate verilog code and C++ simulator framework. Also, for our initial modeling phase, we used a trace based modeling tool TDG and we will be using the same for our further analysis.

For evaluating PENN, we will be collecting performance, power and area statistics of the synthesized version and compare it with statistics given in the literature of DSA papers.

5. Related Work

Neural Networks and Machine learning have a broad range of applications in media, image analysis [14] and speech processing [6]. There is enormous interest shown by the research community in accelerating Deep Neural Networks [11] and Convolution Neural Networks [12]. DianNao [3], Neural Processing Unit [7] are two of the many interesting DSAs that we have considered so far. Programmable architecture is one of the major goals of our design. Dynamically Specialized Execution Resources [9] is software - hardware co-design used for the reconfigurable hardware. GCC compiler is modified to generate the configuration parameters that are required for the reconfigurable hardware in DySER. The most similar design in terms of microarchitecture is MorphoSys [19]. It also embeds a low power TinyRisc core, integrated with a CGRA, DMA engine and frame buffer. Here, the frame buffer is not used for data reuse, and the CGRA is more loosely coupled with the host core. Specialization Engine for Explicit-Dataflow (SEED) [16] makes use of dataflow analysis at fine grained granularity to achieve higher performance. These algorithms can be used to analyze new algorithms to study feasibility to export them on to PENN.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[2] J. Brown, S. Woodward, B. Bass, and C. Johnson, "Ibm power edge of network processor: A wire-speed system on a chip," *Micro, IEEE*, 2011.

[3] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[4] R. Colwell, "The chip design game at the end of moore's law." Hot Chips, 2013, http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.15-keynote1-Chipdesign-epub/HC25.26.190-Keynote1-ChipDesignGame-Colwell-DARPA.pdf.

[5] R. Courtland, "The end of the shrink," *Spectrum, IEEE*, vol. 50, no. 11, pp. 26–29, November 2013.

[6] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8609–8613.

[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[8] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy efficient computing," *IEEE Micro*, 2012.

[9] ——, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, no. 5, pp. 38–51, 2012.

[10] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA, 2010.

[11] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[12] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8595–8598.

[13] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2015.

[14] V. Mnih and G. E. Hinton, "Learning to label aerial images from noisy data," in *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, 2012, pp. 567–574.

[15] Q. Nguyen, "The linux/risc-v installation manual."

[16] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 298–310.

[17] T. Nowatzki, V. Govindaraju, and K. Sankaralingam, "A graph-based program representation for analyzing hardware specialization approaches."

[18] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, S. W. Keckler, D. Burger, and C. R. Moore, "Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture," in *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 422–433.

[19] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and M. C. Eliseu Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465–481, 2000.

[20] B. Sutherland, "No moore? a golden rule of microchips appears to be coming to an end." The Economist, 2013, http://www.economist.com/news/21589080-golden-rule-microchips-appears-be-coming-end-no-moore.

[21] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO*, 2003.

[22] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 249–260. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485944