An Efficient Architectural Realization of a Specialization Engine for Neural Networks With 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. This project focuses on building one such programmable accelerator for neural network domain. We mainly build upon the prior work and preliminary modeling results done for this specialization engine based on specialization principles it employs[1].

For this course project, we targeted deep neural network domain and built a specialization engine called Programmable Engine for Neural Networks (PENN). It should be able to execute variety of deep neural network applications like convolution, classification and pooling kernels. The project focuses on: i) Exploring the trade-offs of many fine-grain design decisions for PENN based on a thorough analysis of neural network workloads, ii) End-to-end functional and timing model implementation of PENN using CHISEL [3], iii) Building complete software toolchain to compile the programs and configure PENN. We have evaluated our design for power and area with state-of-the art neural network DSA – DianNao [5]. We have limited our study only to deep neural networks for this project.

1. Introduction

Diminishing gains of transistor scaling [6, 22, 7] 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 [5, 16, 24, 4]. 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 [10], Wavescalar [23], TRIPS [20] 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) exploiting five specialization principles most of the neural network workloads employ. The main advantage PENN adds is the programmability feature which are missing in most of the DSAs. We explain the specialization principles commonly found in DSAs and employ the same for PENN and derive at the architecture. 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

[1] The modeling and specialization principles are explained in detail in upcoming HPCA 2016 paper

identified and realized in the PENN architecture. Our insight is that PENN could achieve the performance of a DSA specific to each application while maintaining the programmability. PENN also provides a flexible programming interface used to program neural networks. PENN can be configured at synthesis level as well as at runtime based on workload property and configuration provided.

At a high-level, this project makes following contributions:

- Exploring the trade-offs of many fine-grain design decisions for PENN based on a thorough analysis of neural network workloads
- An end-to-end implementation of PENN engine along with well-defined software programming interface
- A framework involving ISA simulator, assembler, scheduler and flexible hardware definition for mapping any neural network program and also any future application domain
- A preliminary area and power analysis compared to state-of-the art DNN accelerator DianNao

The entire project report is organized as follows:
Section 2 explains the prior work done on which this project is defined and built on. Section 3 explains configuring PENN and Section 4 explains the software interface exposed. Section 5 has the details of PENN architecture. Section 6 has PENN framework and implementation details. Results are presented in Section 7 and we discuss related work in Section 9. We also have future plans of extending the project and is explained in Section 8. Challenges faced are also added in the same section and finally we conclude in Section 10.

2. Prior Work

We build our project on the prior work done on LSSD (Work done in Vertical Research Group and to be appeared in HPCA 2016), which includes identification of specialization principles and proposal of the high-level PENN architecture. Before explaining the high level organization of PENN architecture, the five specialization principles employed to consider this style of architecture are explained. The primary insight on coming to the architectural substrate is based on well-understood mechanisms of specialization used in DSAs. The assumptions made for neural network workloads, based on preliminary analysis on Diannao workloads/kernels are explained.

2.1. Workload Assumptions

Before explaining the specialization principles, below are some common workload properties which suits well for PENN architecture. 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.

2.2. Specialization Principles

Broadly, these principles are seen in as a counterpart to the insights from Hameed et al. [12], 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. The former one is chosen 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.

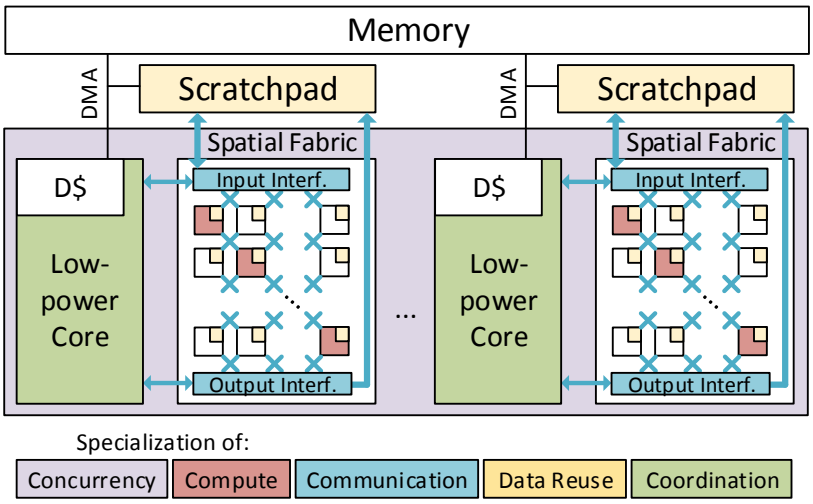


Figure 1: Programmable Engine for Neural Networks (PENN) Architecture

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.

2.3. PENN Overview

Figure 1 gives a high level overview of the PENN architecture 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 straight-forwardly 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

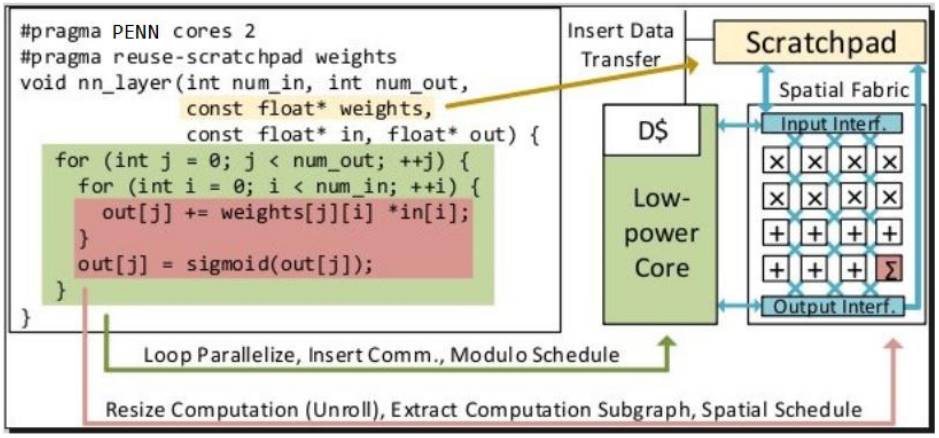


Figure 2: Example Program & Compiler Passes (Arrows labeled with required compilation passes.)

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, an efficient mechanism is required for *coordinating* the above hardware units (e.g. configuring the spatial architecture or synchronizing DMA with the computation). Again, here relying on the simple core is proposed, 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.

3. Configuring and Programming PENN

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

For our project, though many optimization strategies are possible, we have considered the primary constraint of the 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 have taken several micro-architectural design decisions to make the 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.

But in our project, we have developed APIs or MACROs for PENN programming and hand-generate

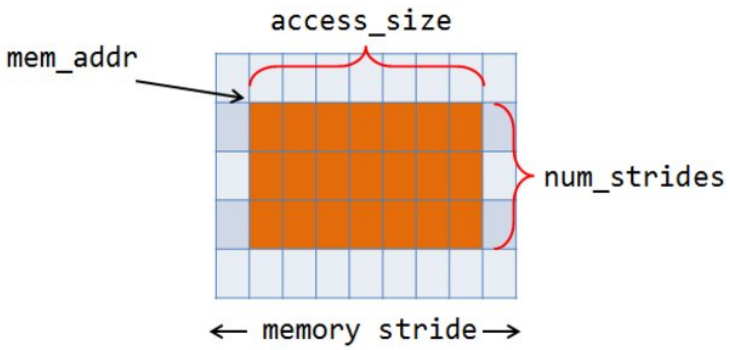


Figure 3: Memory stride access interface for all the kernels

assembly instructions along with configuration stream to configure PENN. We will be explaining in detail about the Software Programming Interface developed in Section 4.

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 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.

4. Programming Interface

In this section, we explain about the software programming interface PENN employs and abstraction provided to programmers. of each hardware component and infrastructure development. Our project's main aim is to develop a hybrid execution model of dataflow and vector processing programmable accelerator. Therefore, hardware-software interface is an important piece to support general purpose programmability along with efficiency.

Figure 3 shows the software interface exposed to programmers for accessing memory for all the kernels. Layout of data structure in the PENN hardware is transparent to the programmer. Neural networks generally have a 3D data structure for convolutional and pooling layers. To support this access pattern, we provide certain abstractions to the programmer. The data structure can be accessed with two dimensional *memory_stride* over *num_strides* iterations. *memory_stride* is the total size of 2D data structure accessed and *num_strides* is the total number of strides required to access all elements in 3D data structure. With respect to the each kernel parameters which are used in all the application kernels we consider (explained in Figure 11) the programming interface is explained below:

Given a neural network with parameters:

N_i: Number of input neurons
N_o: Number of output neurons
N_x: Size of neuron in X dimension
N_y: Size of neuron in Y dimension
K_x: Size of kernel in X dimension
K_y: Size of kernel in Y dimension

Programming interface can be defined as follows:

```
#ifndef penn_INSTS_H
#define penn_INSTS_H

//Stream in the Config
#define DMA_penn_CONFIG(mem_addr, size)

//Fill the scratchpad from DMA.
#define IC_DMA_SCRATCH_LOAD(mem_addr, mem_mem_stride, access_size, num_mem_strides, scratch_addr, word_offset)

//Read from scratch into port_interface.
#define IC_SCRATCH_READ(start_addr, mem_stride, port_type, port_num)

//Read from DMA to wide port interface
#define IC_DMA_READ(mem_addr, mem_stride, access_size, num_strides, port_type, port_num )

//Read from CGRA to Ouput buffers
#define OC_BUF_WRITE(port_type, port_num)

//Write to DMA from OC
#define OC_DMA_WRITE(port_type, port_num, access_size, num_strides, mem_addr, mem_stride)

//Write to scratch from OC.
#define OC_SCRATCH_WRITE(port_type, port_num, scratch_addr, word_offset)

//Write to CGRA back .
#define OC_RECURRENCE(out_port_type, out_port_num, in_port_type, in_port_num, num_strides)

#endif
```

Figure 4: Macros to be included in all PENN programs

mem_addr: Start address of the data structure
memory_stride: N_i x N_x
access_size: N_i x K_x
num_strides: K_y
port_type: Architecturally exposed port type
port_num: Wide port number

The programming interface is used as a set of macros/APIs. Now, with this programming interface exposed, any new neural network program can be written by instantiating set of macros/APIs in a program. Figure 4 gives a high level overview of macros used in a neural network program. Each of the macros implement their some functionality of PENN architecture and will be explained in detail below. Each of these macros are assembled into in-line assembly or intrinsics and are embedded in a regular general purpose program written in a high level language like C, C++. The program with these macros get compiled to low-level PENN assembly instructions which are explained in Section 5.8.

Also, shown below in Figure 5, is an example classifier program written using macros of PENN. This includes macros from CGRA configuration to data movement explained below.

The macros implemented for PENN are explained below:

1. **Input Controller and DMA macros**

   (a) `SET_MEM_STRIDE [mem_addr] [mem_stride]`
   Command to input controller (IC) to set a memory stride starting at this address for further operations. IC uses special registers inside it (M0-M3) to set each memory stride, to track all the following commands of stride access to this particular memory stride. To account for out-of-boundary check, the IC should take care of max-end-address DMA can access for the given stride.

   (b) `IC_DMA_SCRATCH_LOAD [mem_addr] [access_size] [num_strides] [scratch_addr] [word_offset]`
   Command to input controller to stream in certain memory content from global memory and load to scratchpad. (This in turn commands DMA). The stride size must be a multiple of scratch_line size. If it is less than scratch_line size, it is zero padded.

```
// Stream in CGRA config (do this somewhere else?)
int *cgra_config;
int cgra_cofig_sz;
DMA_SB_CONFIG(cgra_config, cgra_config_sz);

// Stream in inputs to scratch
VTYPE neuron_i_scratch[Ni];
neuron_i_scratch = SCRATCHSTART;
IC_DMA_SCRATCH_LOAD(&neuron_i, sizeof(VTYPE), sizeof(VTYPE), Ni, neuron_i_scratch);

// Handle class3, Nn = 32
int pipedepth = PIPEDEPTH;
if(Nn < PIPEDEPTH * 2){
  pipedepth = Nn/2;
}

// Each pipe calculates PIPEDEPTH outputs at once
// This will iterate thru twice for class1, No = 128
for(int n = 0; n < Nn; n += pipedepth * 2){
  for(int i = 0; i < Ni; i+= PIPEWIDTH){
    // Enable sigmoid on final itr
    if(i + PIPEWIDTH < Ni){
      IC_CONST(INPUTPRED0, 0, pipedepth);
      IC_CONST(INPUTPRED1, 0, pipedepth);
    } else {
      IC_CONST(INPUTPRED0, 1, pipedepth);
      IC_CONST(INPUTPRED1, 1, pipedepth);
    }

    // Read in PIPEDEPTH copies of neurons to both pipes
    for(int j = 0; j < pipedepth; ++j){
      IC_SCRATCH_READ(&neuron_i_scratch[i], sizeof(VTYPE)*PIPEWIDTH, INPUTNEURON0);
      IC_SCRATCH_READ(&neuron_i_scratch[i], sizeof(VTYPE)*PIPEWIDTH, INPUTNEURON1);
    }
    // Read in synapeses to both pipes
    IC_DMA_READ(&synapse[n][i], sizeof(VTYPE)*Ni, sizeof(VTYPE)*PIPEWIDTH, pipedepth, INPUTWEIGHT0);
    IC_DMA_READ(&synapse[n+pipedepth][i], sizeof(VTYPE)*Ni, sizeof(VTYPE)*PIPEWIDTH, pipedepth, INPUTWEIGHT1);

    // Pass accumulated sums back except on last itr
    if(i + PIPEWIDTH < Ni){
      OC_RECURRENCE(OUTPUT0, INPUTACC0, pipedepth);
      OC_RECURRENCE(OUTPUT1, INPUTACC1, pipedepth);
    }
  }
  // write completed outputs out to memory
  OC_DMA_WRITE(OUTPUT0, sizeof(VTYPE), sizeof(VTYPE), pipedepth, &neuron_n[n]);
  OC_DMA_WRITE(OUTPUT1, sizeof(VTYPE), sizeof(VTYPE), pipedepth, &neuron_n[n + pipedepth]);
}
```

Figure 5: Classifier Program written using PENN Macros

(c) `IC_SCRATCH_READ [scratch_addr] [word_offset][port_type] [port_num]`
Command to IC for loading data from scratchpad to vector/scalar ports. Our design has variable number of vector and scalar ports. The vector and scalar ports are numbered and is exposed to the SODOR core. Vector port is 64B wide to match the scratchpad line size and scalar port is 16bits (2B or 1 neuron size). There is also a field in the command to specify the exact line address to be read. By default we start reading from the base address 0, and the IC keeps track of how many lines have been read. For scalar ports we need the word offset into scratchpad line to stream in the exact 16bit neuron or partial sum. When reading to vector ports this word_offset will be set to zero.

(d) `IC_DMA_READ [start_addr] [access_size] [num_strides] [port_type] [port_num]`
This macro is used to stream in the data directly from DMA engine into wide port interface as in the case of streaming synaptic weights.

(e) `SET_ITER [iter_num]`
We need a command from core to IC, to perform certain operations for specific number of iterations. This macro configures the *iter_reg* register inside the IC and is used for synchronization purposes.

(f) `IC_CONST [const_value][num_strides]`
This macro is used when the input controller needs to stream in constant values for *iter_reg* cycles.

2. **Output controller and DMA macros**

   (a) `OC_SB_READ [port_type] [port_num]`
   This macro is used to stream in the data to output buffers from DMA engine or scratchpad.

   (b) `OC_SB_SCRATCH_WRITE [port_type] [port_num][scratch_addr] [word_offset]`
   For a large neural network and operation, we might store the results in scratchpad directly (from both vector and scalar ports). We make use of this macro in such scenarios.

   (c) `OC_BUF_DMA_WRITE [mem_addr] [mem_stride] [access_size] [num-strides]`
   We use this macro to stream in the data from output buffers to memory via DMA. For writing it to DMA, we always store the data to the buffers and stream out of the buffers. OC coalesces the buffered values as 64B data writes.

   (d) `OC_SB_WRITE [port_type] [port_num] [port_type] [port_num]`
   We use this macro when we want to directly use the reduced output values into CGRA input port interface. The out port and in port type should match for this case.

5. PENN Architecture

In this section, we explain the PENN architecture and design decisions taken in detail. Figure 10 shows the detailed architecture of PENN with individual modules as explained below. We start from the in-order SODOR core, controllers for data movement into CGRA and also explain the re-use hardware structure scratchpad. We also list out module level hardware interfaces for some important modules (controllers) and design decisions taken.

5.1. SODOR Core

We make use of 3 stage pipelined RISC-V processor core developed by UC Berkeley as our low power core. We have extended the available Scala source code for SODOR core to recognize PENN instructions listed in Section 5.8 in decode stage and dispatch them to PENN Command Queues in the required format. We have added an interface between the decode stage of SODOR core and PENN Command Queues.

5.2. Direct Memory Access (DMA)

DMA is used to move the data to and from memory and scratchpad. DMA interacts with input controller (IC), output controller (OC), scratchpad memory and main memory. DMA receives the commands from both IC and OC to load and store the memory strides to/from memory. Once DMA receives the command, it performs actions required and sends request complete signals back to input/output controller.

5.3. Scratchpad

Scratchpad is a specialized memory used to store the reusable data for CGRA fabric. Scratchpad can receive data from either DMA or output buffers. The input controller instructs the scratchpad to write into the port, which will be used for computations. DMA also reads from and writes to the scratchpad as required depending on the instructions from the input controller. Each line of scratchpad is of length 512 bits (64B). 32 input neurons of two bytes each can be read in one cycle. Since we are considering 4 wide SIMD functional units, each scratchpad line can feed 8 input ports simultaneously.

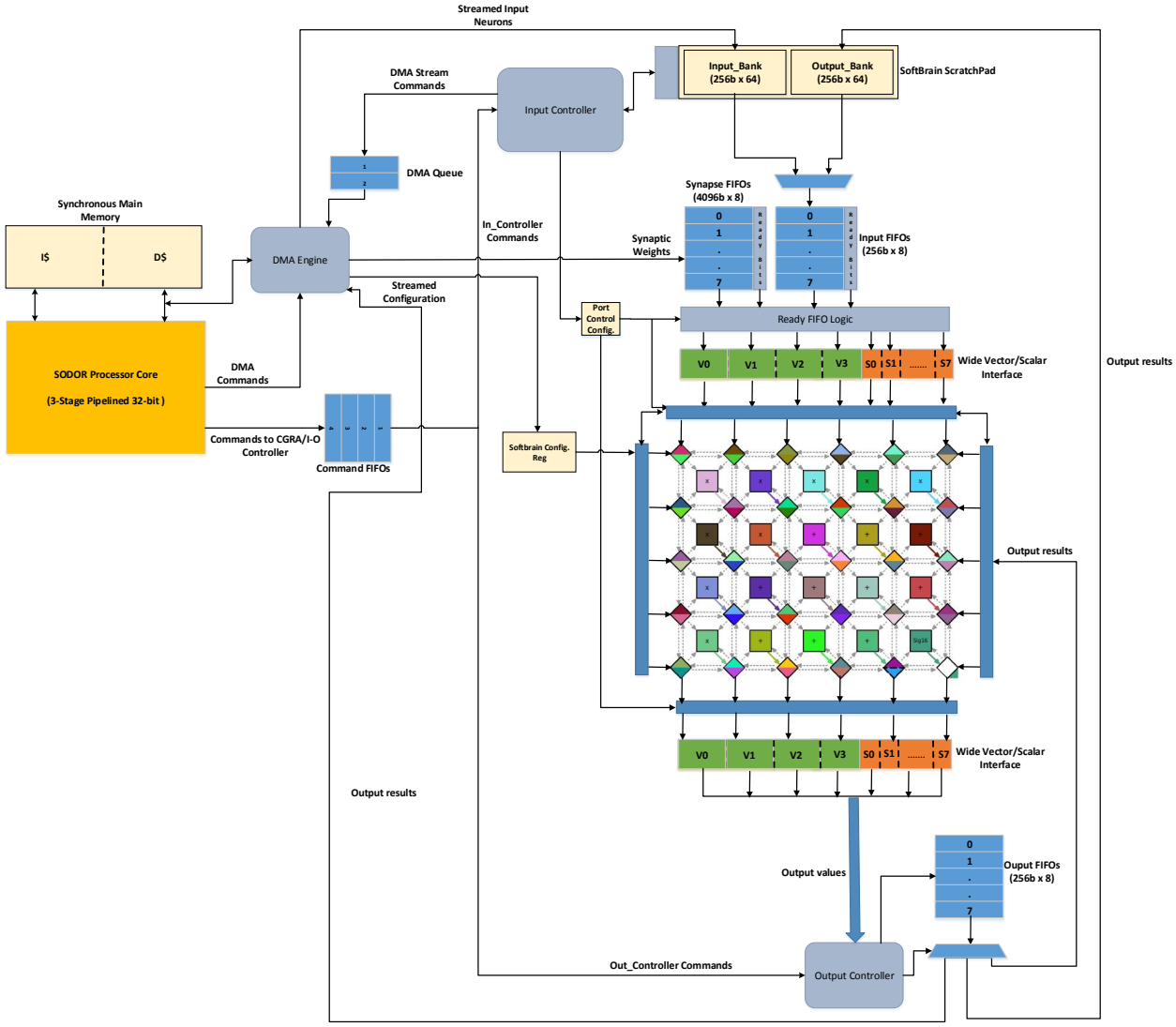


Figure 6: PENN Detailed Architecture

5.4. Input Controller:

The input controller receives CGRA instructions from SODOR core via PENN input command queue. It co-ordinates with DMA, output FIFOs and scratchpad for data movement. It also sends signals to indicate configuration mode, stall etc. Port interface of input controller is as follows:

```
module input_controller (
  // Command Queue and SODOR Core Interface
  input cq_ic_instr,
  input cq_ic_ready,
  output cq_ic_read_instr,
  output ic_cq_readdone,

    // DMA Interface
  input dma_ic_ready,
```

```
    output ic_dma_command,
    output ic_dma_valid,

      // Scratchpad Interface
    input scratch_ic_ready,
    output ic_scratch_addr,
    output ic_scratch_valid,
    output ic_scratch_rbar_w,
    output ic_scratch_numstrides,
    output ic_scratch_portinfo,
    output ic_scratch_wordoffset,

      // Port Interface
    output ic_port_portinfo,
    output ic_port_value,
    output ic_port_valid
)
```

5.5. Port Interface:

Port interface is inspired from the previously proposed spatial architecture fabrics. It provides the flexibility to distribute the data from the scratchpad line to any input CGRA port. There are 2 types of ports – Scalar and Vector. Scalar ports are used to store partially computed output sums. Vector ports are used to hold wide data sizes upto 64bit neurons and synapses. The port interface to CGRA input ports can be configured during the configuration stage. The ready logic monitors the data arrival at port interface and feeds the data to CGRA input ports. It also sends back-pressure signals when no more input data can be consumed.

5.6. Output Controller:

The output controller receives CGRA instructions from SODOR core via PENN output command queue. It is responsible for moving the data from the output FIFOs to scratchpad, DMA and input port interface. Port interface of Output Controller is as follows:

```
module output_controller (
  // Command Queue and SODOR Core Interface
  input cq_oc_instruction,

    input cq_oc_valid,
  output oc_cq_ready,

    //CGRA Output Interface
  input port_oc_data,
  input port_oc_valid,

    //Output Buffer Interface
  output oc_obuf_data,
  output oc_obuf_valid,
```

```
    //DMA Interface
    output oc_dma_command,
    output oc_dma_valid,
    input dma_oc_ready,

    // Scratchpad Interface
    output oc_scratch_addr,
    output oc_scratch_data,
    output oc_scratch_numstrides,
    output oc_scratch_wordoffset,
    output oc_scratch_we,

    // Port Interface
    output oc_port_info,
    output oc_port_data,
    output oc_port_we,
)
```

5.7. CGRA Fabric

Course Grained Reconfigurable Array (CGRA) Fabric consists of matrix arrangement of switches and functional units. The fabric can be configured to route the data as required depending on the workload. Our design contains `6 x 5` switches with `5 x 4` functional units. These numbers are parametrized, so it is possible to create fabric of any dimension. Figure 7 shows the high level details of the 1 CGRA tile in the fabric which includes the PENN_Switch and a Functional Unit (FU).

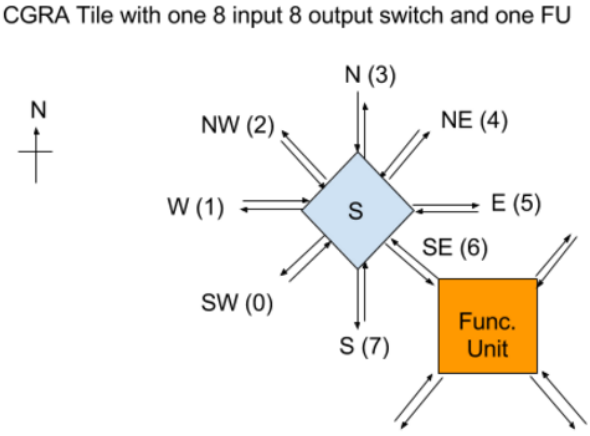


Figure 7: 1 CGRA Tile with 8 inputs and 8 outputs

PENN Switch: Each PENN Switch consists of six input ports, eight output ports, eight 3bit `cfg_reg` to direct each input to one of the output ports and eight `8:1` mux, one at each output port.

Data width is parameterized to support varying SIMD length and number of input/output ports are also configurable. We use the parameter only to study the power and area variations of the switches for different number of inputs and outputs.

Functional Unit: Each FU can handle 4 SIMD width operation.

1. *Adder FU:* There are two configurations within adder FU. This configuration is done during `cfg_mode`. Since each FU has to support SIMD operation, adder FU contains 4 16bit adders. In first configuration, 4 adders within FU perform four separate additions. In the second configuration, three adders are used for `4:1` reduction and fourth adder as accumulator.

2. *Multiplication FU:* Four multipliers within FU performs four multiplications.

3. *Sigmoid FU:* If predicate bit is set, it performs sigmoid operation. If this bit is not set, it just passes the result and acts as a bypass stage. Each FU can receive inputs from four different ports. But only two of them will be used. In order to select two from four, each FU has two 2bit configuration registers namely `cfg_in0` and `cfg_in1` whose values are set during cfg_mode.

CGRA Configuration: CGRA is configured using configuration stream generated from the scheduler. Figure 8 shows the configuration stream sent to CGRA Fabric. This includes select line bits for the muxes at each outport port as well as opcode, output direction of functional units. Th input controller sets *cfg_mode* signal to CGRA and the bit stream is sent to all the tiles of CGRA in five cycles.

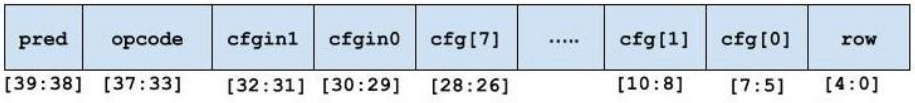


Figure 8: CGRA Configuration Bitstream

5.8. PENN Instructions

As discussed in Section 4, the high level macros need to be assembled into low level instructions. Since we have chosen SODOR core as the low power core, we extended the RISCV base user integer 32bit ISA [1] to include the new PENN instructions.

The opcode space of 32bit encoding space of RISCV ISA can have 4 possible encodings with last 2 bits always as `11`. The bits `4:2` can have 2 possible encodings for custom instructions – `010, 110`. The first 2 bits can be `00,01` for `010` encoding and `10,11` for `110`. So with these 4 major opcodes, we assigned each of them for the major operation of PENN. And then, we make use of the 3bit `funct` in the encoding to have more instructions in each major type. The four major type of opcodes are summarized in Table 1:

Opcode [6:0]	Major Operation
00_010_11	DMA
01_010_11	Input Controller (IC)
10_110_11	Output Controller (OC)
11_110_11	Misc/Synchronization

Table 1: Major Opcodes of PENN

All the PENN instructions follow 2 types of instructions: First one is `I-Type` which is an immediate type instruction and second one is `S-Type` which has split immediate field to support smaller immediate values. Figure 9 shows the 2 formats used for all the instructions.

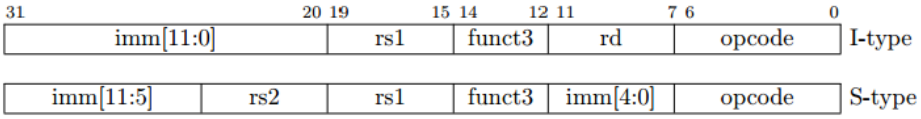


Figure 9: PENN Instruction Formats

Below is the detailed listing of all the instructions encoded for the PENN macros. The macros get assembled to these instructions and then gets decoded as the PENN related instructions in SODOR Core.

1. DMA_Type (DMA Config): Instruction to configure PENN with bits streamed from memory
   `penn_cfg rs1, imm[11:0] -- I_TYPE`
   ```
   rs1 = mem_addr
   imm = size in bytes
   ```

2. Misc: Instruction to set the memory stride for a given iteration
   `penn_stride rs1, rs2 -- S_TYPE`
   ```
   rs1 = mem_addr
   rs2 = mem_stride
   ```

3. Misc: Instruction to load the memory address and access size values
   `penn_dma_addr rs1, rs2 -- S_TYPE`
   ```
   rs1 = access_size
   rs1 = mem_stride
   ```

4. IC_Type (Scratchpad load): Instruction to load the values to scratchpad
   `penn_dma_scr rs1, rs2, imm[5:0] -- S_TYPE`
   ```
   rs1 = num_strides
   rs2 = scratch_addr
   imm[5:0] = word_offset
   ```

5. IC_Type (Scratchpad read): Instruction to read the scratchpad to port interface
   `penn_scr_rd rs1, rs2, imm[11:0] -- S_TYPE`
   ```
   rs1 = scratch_addr
   rs2 = num_strides
   imm[4:0] = port_num
   imm[5] = port_type
   imm[11:6]= word_offset
   ```

6. IC_Type (DMA read): Instruction to load values to port interface from DMA
   `penn_dma_rd rs1, imm[5:0] -- I_TYPE`
   ```
   rs1 = num_strides
   imm[4:0] = port_num
   imm[5] = port_type
   ```

7. IC_Type (Stream Constant): Instruction to IC to stream in a constant value for stride number of times
   `penn_const:  rs1, rs2 imm[11:0] -- I_TYPE`
   ```
   rs1 = Constant value
   rs2 = num_strides
   ```

```
imm[4:0] = port_num
imm[5] = port_type
```

8. OC_Type (Read Output from CGRA): Instruction to read the results from CGRA ports to output buffers
   penn_wr imm[5:0] -- S_TYPE
   ```
   imm[4:0] = port_num
   imm[5] = port_type
   ```

9. OC_Type (Write output to scratchpad): Instruction to write value from port interface to scratchpad
   penn_wr_scr rs1, rs2, imm[11:0] -- S_TYPE
   ```
   imm[4:0] = port_num
   imm[5] = port_type
   rs1 = scratch_addr
   rs2 = num_strides
   imm[11:6] = word_offset
   ```
10. OC_Type (Write output to DMA): Instruction to write values from output buffers to DMA
    penn_dma_addr_p rs1, rs2, imm[5:0] -- S_TYPE
    ```
    rs1 = mem_addr
    rs2 = access_size
    imm[4:0] = port_num
    imm[5] = port_type
    ```

11. OC_Type (Write output to DMA): Extension of previous instruction to execute stride number of times
    penn_wr_dma rs1 -- I_TYPE
    ```
    rs1 = num_strides
    ```

12. OC_Type (Write output back to CGRA): Instruction to load values back to input port interface
    penn_wr_rd rs1, imm[11:0] -- S_TYPE
    ```
    imm[4:0] = out_port_num
    imm[5] = out_port_type
    imm[10:6] = in_port_num
    imm[11] = in_port_type
    rs1 = num_strides
    ```

6. Implementation

In this section, we explain PENN implementation in detail. we first ex paling overall framework of how PENN is synthesized and then explain the individual software pieces of the framework.

6.1. PENN Implementation Framework

Figure 10 shows how the entire PENN framework is implemented and evaluated. For a given computation subgraph (kernel) discovered, PENN scheduler will generate the timing schedule for spatial fabric based on some user provided and synthesis time constraints. This configuration file called *penn_config* includes the type of FU modules, a flexible port interface definition and other details. The generated configuration bits from the scheduler are then embedded back in the program and is assembled to generate the PENN binary. This binary is fed to the ISA simulator and is functionally evaluated whether the configuration stage and the instructions are correctly simulated. We have implemented this infrastructure to allow flexibility to add new PENN instructions and make it is correctly getting simulated.

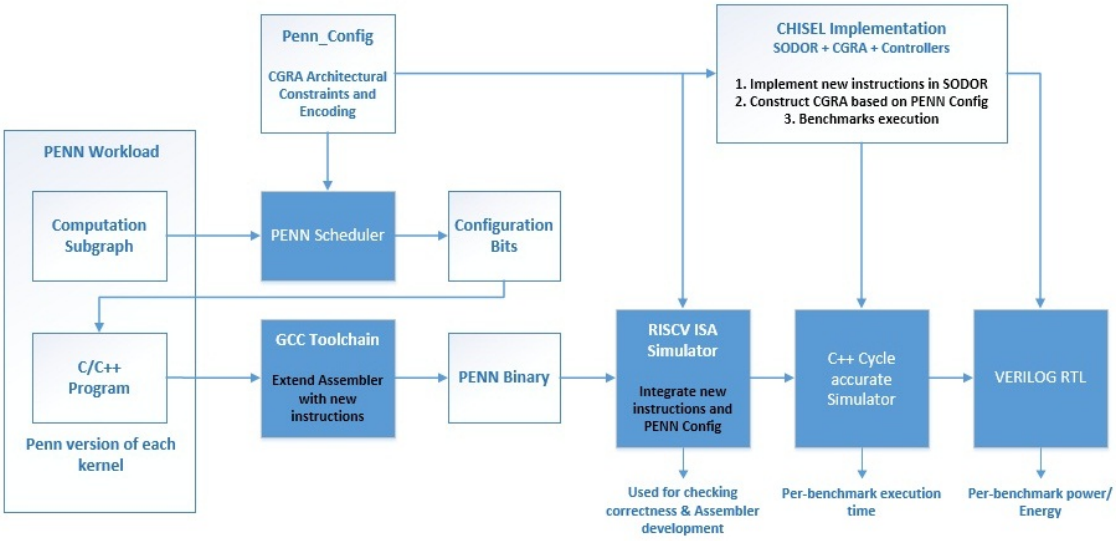


Figure 10: PENN Implementation Framework

The remaining major part is the Chisel implementation of all the modules of PENN. It is implemented based on the module definitions defined in the Section 5. All the individual modules are written and we are testing the system as a whole now. These modules are then integrated with the SODOR core, its memory subsystem and a C++ cycle accurate simulator is generated. The cycle accurate simulator gives us the per-benchmark performance results. We can also generate the Verilog of the entire system along with DMA, Scratchpad and CGRA and get the overall area/power results. Per-benchmark energy can be obtained by getting the activity factors from the cycle accurate simulator and estimating the energy.

The program binaries are loaded into the SODOR memory through an emulator wrapper written for SODOR core and PENN. We make use of RISCV assembler to generate the binary and then use their hex converter to load the memory dump.

6.1.1. Assembler We make use of the existing RISCV GNU toolchain to extend the assembler to compile the programs and generate the assembly instructions. We made use of intrinsics which basically substitutes the inline function with a set of PENN assembly instructions when made aware to the assembler. We have implemented the assembler with all new encodings and the assembler is also extended to generate the configuration bits needed for the CGRA configuration.

6.1.2. ISA Simulator To test the new implemented instructions and the configuration bits obtained from the assembler, we extended the RISCV ISA functional simulator called Spike [2]. The ISA simulator reads in the binary generated from the assembler and then simulates for all the instructions encountered in the binary. To check the correct implementation of ISA, we load pre-determined values to scratchpad and expect to be read the same when a scratchpad read instruction is executed. Similarly, all the other instructions are tested thus giving an idea that instructions are functionally executing correctly.

6.1.3. Scheduler The CGRA in PENN is statically scheduled and this timing based schedule information has to be passed to the CGRA while configuring it. We heavily rely on the Integer Linear Programming (ILP) Scheduler for Spatial Fabric developed by Tony et. al [19]. We modified this spatial scheduling framework to schedule two `16:1` reduction network with multipliers, adders and sigmoid units synthesized in `6 x 5` CGRA.

Layer	N_x	N_y	K_x	K_y	N_i	N_o	Description
CONV1	500	375	9	9	32	48	Street scene parsing
POOL1	492	367	2	2	12	-	(CNN) [13], (e.g.,
CLASS1	-	-	-	-	960	20	identifying "building", "vehicle", etc)
CONV2*	200	200	18	18	8	8	Detection of faces in YouTube videos (DNN) [26], largest NN to date (Google)
CONV3	32	32	4	4	108	200	Traffic sign
POOL3	32	32	4	4	100	-	identification for car
CLASS3	-	-	-	-	200	100	navigation (CNN) [36]
CONV4	32	32	7	7	16	512	Google Street View house numbers (CNN) [35]
CONV5*	256	256	11	11	256	384	Multi-Object
POOL5	256	256	2	2	256	-	recognition in natural images (DNN) [16], winner 2012 ImageNet competition

Figure 11: Benchmark Layers (CONV=convolutional, POOL=pooling, CLASS=classifier, CONVx* indicates private kernels)

6.1.4. Application Kernels For deep neural network kernels, we mainly use the same neural network topology as in DianNao paper and want to limit to these kernels for our study. We have written all these kernels using our hardware-software programming interface explained in the Section 4. The hardware design choices made to support these kernels is explained in Section 5. Figure 11 gives the overview of kernels we will be using for our evaluation and it also lists out the kernel sizes (K_x x K_y) for each neural network program.

7. Evaluation and Results

The entire PENN system integration part is a huge part of the project and is still an on-going work and may take few more weeks to get cycle by cycle analysis of the micro-benchmarks we have written. Even though we have individual software parts ready, the Chisel based hardware modules and SODOR core needs a thorough validation using the PENN instructions we have written. So, as of now we present the preliminary modeling performance results[2]. These performance results serve as a reference to make sure when we have the cycle accurate results we fall in the same ballpark.

But we do have the detailed area and power analysis done by synthesizing the design we obtained from Chisel based implementation. We first present the preliminary modeling performance results and then present the detailed area and power numbers.

7.1. Performance Analysis [Based on Prior Work]

In this section, we compare the performance of DianNao and domain-provisioned PENN design in Figure 12, normalized to a 4-wide OOO core. The In-Order core results were initially obtained by using datasheets of a 3-wide VLIW machine (LX3) but the new results are obtained by extending it to 3-stage pipelined SODOR core. We expect it the a similarly provisioned In-Order core results to be same as SODOR core. To highlight the sources of benefits from each specialization principle in PENN, we also include four additional design points, where each builds on the capabilities of the previous design point. The four design points and SIMD design point are explained below:

1. *Core + SFU:* The In-order SODOR core with added problem-specific functional units (computation specialization).
2. *Multicore:* PENN multitile/core system (+concurrency).

[2]This is based on the prior results which is appearing in HPCA 2016 paper

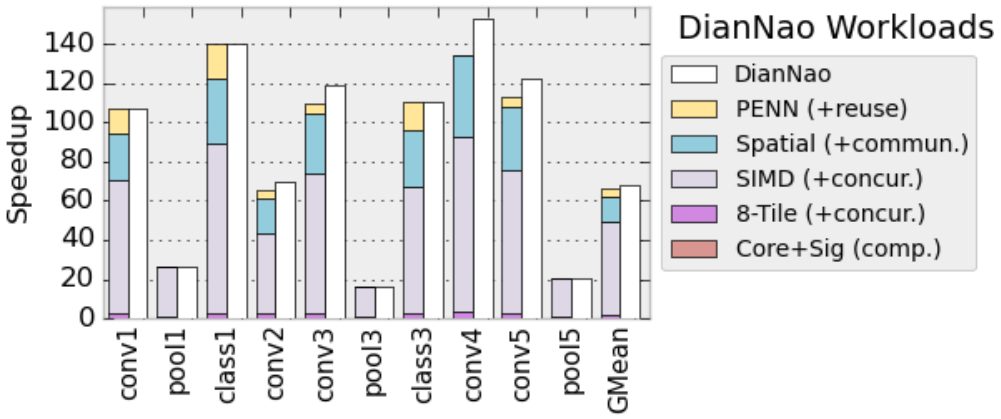


Figure 12: PENN vs DianNao Modeled Performance

3. *SIMD:* PENN with SIMD, its width corresponding to PENN's memory interface (+concurrency).
4. *Spatial:* PENN where the spatial arch. (CGRA) replaces the SIMD units. (+communication).
5. *PENN:* Previous design plus scratchpad (+reuse)

DianNao versus PENN Performance Figure 12 presents the speedup comparison of PENN with DianNao for set of convolutional, classifier and pooling workloads mentioned in Section **??**. PENN's performance is similar to DianNao on most workloads. The In-order core + SFU performs worse than the OOO core and hence the speedup is below 1 in the figure. Concurrency was most important contributor for performance (8× multicore, 32× SIMD). The CGRA provides an additional small benefit by reducing instruction management and hides latency of fetching neurons and streaming them into scratchpad. Adding a reuse buffer reduces cache contention and also provides better latency improvement while processing input neurons; The combined speedup of CGRA + Scratchpad is 35%. For pooling, DianNao and PENN maxed out the memory-bandwidth limit. In the remaining convolutional and classifier workloads, PENN has minor instruction overhead (eg. managing the DMA engine).

Takeaway: PENN design has competitive performance with DianNao. The performance benefits come mostly from concurrency *compared to any other specialization technique.*

When we get the cycle level performance activity, we expect to have some overhead from input and output controller steaming and DMA activity. Optimizing these overhead and hiding the streaming latency is one part which has to be focused once we get the entire system running up.

7.2. Area Analysis

For area analysis, we used the Chisel generate Verilog files and synthesized using the 32nm Free PDK library at the frequency rate of 1GHz. The DianNao area numbers are obtained from the paper and are scaled to 32nm library and 1GHz operating frequency(DianNao implementation is done in 65nm and 800MHz). Figure 13 a) presents the area results of 1 instance (tile) of PENN with CGRA Fabric, In-order core + Cache subsystem, input/output controller and the Scratchpad. The overall area of 1-tile PENN is 0.24mm^2. The major part of area is taken up by the CGRA Fabric and this is expected because of large switches and FUs. Each switch routes a 64-bit data item through all possible 6 directions and need giant multiplexers for them. Even the FUs are 64-bit SIMD datatype and hence consume more area. Table 2 has the area breakdown of 36 CGRA switches and 25 FUs. Scratchpad is a 4-KB SRAM and consumes around 16.67% of overall area.

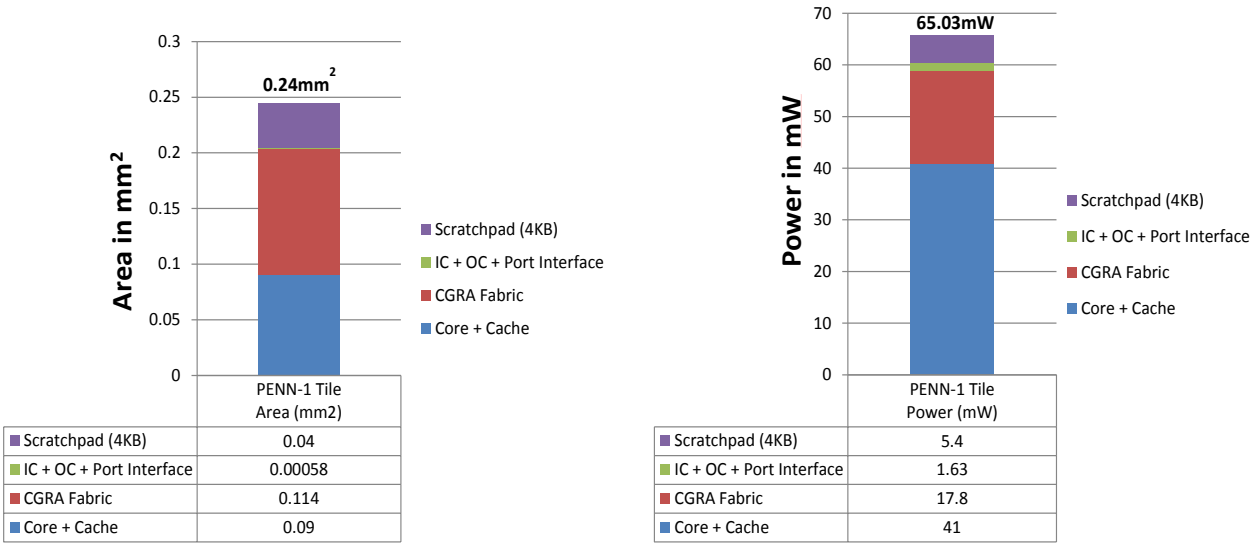


Figure 13: a) PENN Area Results; b) PENN Power Results

Switches		FUs	
Area (mm^2)	*Power (mW)*	*Area (mm^2)*	*Power (mW)*
1.109	10.2	0.0048	7.6

Table 2: Area and Power Breakdown of Switches and FUs of CGRA

7.3. Power Analysis

Power analysis is done similar to area with 32nm technology library and 1GHz operating frequency. Even though we do not consider per-benchmark activity factor based power estimation, since most of the kernels have speedup in the similar range we expect the per-benchmark activity factor to be same and hence the power consumption. However, this has to be evaluated again once we get the cycle level activity infrastructure working. Figure 13 b) presents the power results of 1-Tile PENN. The In-order core + cache subsystem consumes around 64% of total PENN power and this is because its a general purpose core.CGRA fabric consumes 27% of total power and this can be reduced by power gating some FUs when not used. Input/Output controller and scratchpad consumes around 10% of PENN power which is not significant. There is lot of scope to reduce power consumption in CGRA fabric with some careful design optimization decisions. Table 2 has the power breakdown of 36 CGRA switches and 25 FUs.

7.4. PENN vs DianNao Area and Power Comparison

Area (mm2)		Power (mW)	
8-Tile PENN	*DianNao*	*8-Tile PENN*	*DianNao*
1.96	0.56	526.64	213

Table 3: PENN vs DianNao Area and Power Comparison

Finally, we compare the area and power of PENN with the DianNao overall power and area. Table 3 shows the comparison of 8-tile PENN and DianNao area and power. Number of PENN tiles was chosen to be 8 to fit all the choosen deep neural network kernels. 8-tile PENN has 3.5x area and 2.5x power overhead. The area overhead is mainly from multiple instances of CGRA network and power overhead is due to the presence of 8 instances of in-order SODOR core.

Takeaway: PENN design has 3.5x area and 2.5x power overhead compared to the DianNao design. This overhead gets amortized if more domains like CNN, Regex matching or file compression are mapped to same PENN engine.

All the results presented above are preliminary results and we expect once we have the full system integration done and have support for a full benchmark to analysis workflow, more accurate results can be obtained.

8. Challenges and Future Work

The biggest challenge we faced in the project is to figure out how to proceed with this hardware-software design approach. It was fortunate that we came across the RISCV toolchain and the Chisel tool for the end-to-end implementation of PENN. Other challenges we faced were, deciding on hardware interface for the corresponding software abstraction exposed in the program. This would have been very difficult without Chisel, to modify the hardware every-time a new abstraction is put out. Chisel enables to have the *synthesis* level configuration possible for PENN with flexible hardware modification interface. We also learned a lot while working on the assembler, scheduler and instruction-level simulator about how to go on and build such a hardware-software co-design programmable accelerator.

The important work to be done as a first to step is to finish the system integration part which is still a major part. We will also be working on some software pieces which still needs tuning based on the abstractions we have added. We also need to work on micro-benchmarks and see if there are any missing pieces that the software interface does not provide. We also need to explore on whether it makes sense to go for a cache based design for PENN or still rely on the DMA + Scratchpad strategy. On the side of application domains, there are domains which have similar workload properties and could make use of the concurrency PENN hardware provides as well as the programming abstractions the software interface exposes. These include database processing, regex parsing, file compression etc. PENN could be a good hardware substrate for these application domains, and when mapped we could reduce the overall area and power overhead as all these domains would share the same hardware structures.

9. Related Work

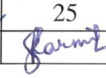
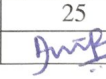
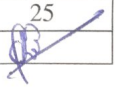
Neural Networks and Machine learning have a broad range of applications in media, image analysis [17] and speech processing [8]. There is enormous interest shown by the research community in accelerating Deep Neural Networks [13] and Convolution Neural Networks [15]. DianNao [5], Neural Processing Unit [9] are two of the many interesting DSAs that tagret deep neural networks. Programmable architecture is one of the major goals of our design. Dynamically Specialized Execution Resources [11] 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 [21]. 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) [18] 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. Spinnaker [14] is one such work which tries to add programmability to the hardware design executing DNNs by using large number of cores.

10. Conclusion

This project proposes a Programmable Engine for Neural Networks (PENN) targeting the DSA efficiency while maintaining the programmability. We mainly target deep neural network (DNN) domain and built a specialization engine employing five specialization principles. PENN can be configured at synthesis level as well at runtime based on the configuration provided. The project focuses on an end-to-end implementation of

PENN along with the support of RISCV software toolchain. PENN provides a flexible software interface to programmers able to program any new neural network. The software interface is exposed through a set of macros which get assembled to low-level PENN instructions. We evaluate PENN's area and power to stat-of-the-art DNN DSA DianNao. In overall PENN has similar performance to DianNao and has 3.5x area and 2.5x power overhead. We believe these overheads can be amortized when new application domains are mapped to PENN. This project also shows that you can still be able to get the efficiency of a DSA by having flexibility through programmability.

11. Statement of Work

Milestones	Vinay	Sharmila	Anilkumar	Chandana
Brainstorming and Design Decisions	25	25	25	25
Low Power Core Integration	80	0	20	0
RISCV Toolchain	50	0	0	50
CGRA Implementation	0	80	0	20
Input/ Output Controller	0	50	0	50
DMA Engine/ Scratchpad Memory	0	20	80	0
ISA Simulator and Assembler	50	0	50	0
Results and Analysis	25	25	25	25
Report Writing	25	25	25	25
Signature				

12. Acknowledgements

We would like to thank Prof. Mikko Lipasti for allowing us to work on this project and also be a guide and provide feedback throughout the course. We would also like to thank Tony Nowatzki and Prof. Karu Sankaralingam for allowing us to take up their research work and extend it to suit our project needs/goals.

References

[1] "Riscv 2.0 isa specification." [Online]. Available: http://riscv.org/spec/riscv-spec-v2.0.pdf

[2] "Riscv isa simulator – spike." [Online]. Available: https://github.com/riscv/riscv-isa-sim

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. Ieee, 2008, pp. 2849–2856.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 495–506, 2013.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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