Cherry Instruction Set (cisa)

16 bit instructions. 5 bits to decide which instruction to use. Remaining bits for parameters.

4 registers: input, weights, acc, out. Each holds a small matrix tile (4x4, 16x16, 32x32 depending on cherry device)

Arithmetic Instructions

Instruction	Example	Meaning	Comments
Matrix Multiply	`matmul`	REG[ACC] += REG[INPUT] @ REG[WEIGHTS]	
Multiply Accumulate	`mulacc`	REG[ACC] += REG[INPUT] * REG[WEIGHTS]	Element wise multiply
Vector Add	`add`	REG[OUTPUT] = REG[INPUT] + REG[WEIGHTS]	
Vector Subtract	`sub`	REG[OUTPUT] = REG[INPUT] - REG[WEIGHTS]	
Vector Multiply	`mul`	REG[OUTPUT] = REG[INPUT] * REG[WEIGHTS]	
Vector Divide	`div`	REG[OUTPUT] = REG[INPUT] / REG[WEIGHTS]	
Vector Power	`pow`	REG[OUTPUT] = REG[INPUT] ** REG[WEIGHTS]	
Vector Reduce Max	`max $1 $2 $3`	max([REG[ACC] if $1 else none, (REG[OUT]]). $2 is axis, $3 is keep dims	Gets the max value in out reg or out reg and acc reg
Vector Reduce Sum	`sum $1, $2, $3`	sum([REG[ACC] if $1 else none, (REG[OUT]]), $2 is axis, $3 is count	Gets the sum of out reg or out reg + acc
Vector ReLU	`relu`	max(x,0) of every x in REG[INPUT]	
Vector Exp	`exp`	e^x of every x in REG[INPUT]	
Vector Log	`log`	log_2 of every x in REG[INPUT]	
Vector GT0	`gtz`	1 if x > 0 else 0 of every x in REG[INPUT]	
Copy ACC	`copy_acc_to_output`	REG[OUTPUT] = REG[ACC]	
Zero ACC	`zero_acc`	REG[ACC] = 0	

Data Transfer Instructions

Instruction	Example	Meaning	Comments
Load	`load $1, $2, $3, $4, $5, $6`	REG[$2] = ... $3 and $4 are lengths. $5 is zero flag, $6 is skip flag. APU $1	Uses address from an APU. Cache to registers.
Store	`store $1, $2, $3, $4, $5`	MEM[addr] = REG[$2][:$3][:$4] and APU $1. If $5 then skip local cache and DMA. Useful for when you need a value for backward pass but wont read it again from cache.	Use address from an APU. Registers to cache.
Mem Read	`mem_read $1, $2, $3`	$1 and $2 are cache addr $3 is APU	ram to cache
Mem Write	`mem_write $1, $2, $3`	$1 and $2 are cache addr $3 is APU	cache to ram

Control Flow Instructions

Instruction	Example	Meaning	Comments
start independent loop	`start_independent_loop ($1)`	start loop with ro_data[$1] params	From ro_data registers gets loop iteration count and number of instructions in the loop body. Loop iterations can be reordered freely. must be no loops nested inside this loop
start loop	`start_loop ($1)`	start loop with ro_data[$1] params	This loop must be executed in order. We can’t rearrange when different loop bodies run. This is low performance loop
end loop	`end_loop_or_jump_if_needed ($1)`	Gets jump number from ro_data[$1]	jumps if loop has more more iterations remaining

Read only data (program header)

Not all instructions can access all registers. Furthermore, read-only data is considered ro_data compiled into the kernel. This header section is 1212 bits.

- 8 loops allowed
  - Each loop needs 15 bits for the number of iterations
- 4 cisa_load/cisa_store address formulas allowed
  - Each address is a linear equation with 8 coefficients (gradients w.r.t. each loop variable) and an offset. Each value is 15 bits so one linear equation needs 135 bits
  - 30 bits for hard coded stride_x and stride_y
- 4 cisa_mem_* address formulas allowed
  - Same as cisa_load/cisa_store but now our addresses are bigger so the offset needs to be 18 bits instead of 15. So each linear equation needs 138 bits.

Thoughts

A program has 1332 bit ro_data in its header. On average 160 bits body for 10 instructions. So each program is 186 bytes.

128KB icache is 686 programs. A program is only valid for certain input tensor shapes and input tensor memory locations (cache or device memory)

Variable sized program headers can decrease pcache transistor usage.