IMAGine Instruction Set

MD Arafat Kabir

May 14, 2024

1 IMAGine Architecture

This is a very brief introduction to IMAGine's high-level architecture. IMAGine is a GEMV accelerator consisting of two submodules: the GEMV array and the column-shift-registers. In Fig. 1 (a), shows these modules. The interface to IMAGine are two FIFOs. You push your instructions through the FIFO-in, and IMAGine pushes out the output through the FIFO-out. The output needs to be stored in the column-shift-registers before they can be pushed to the FIFO-out.

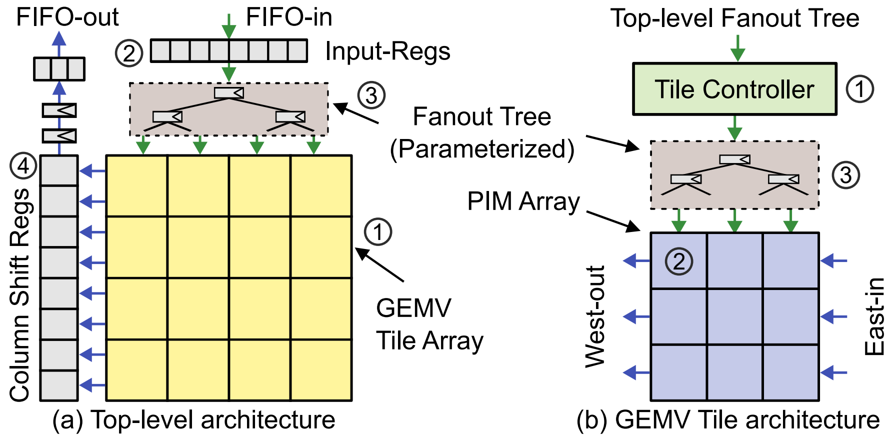

Figure 1: System architecture of IMAGine (a) GEMV engine (`mv_`) & column-shift-register (`vv_`) submodules, (b) Architecture of a GEMV tile.

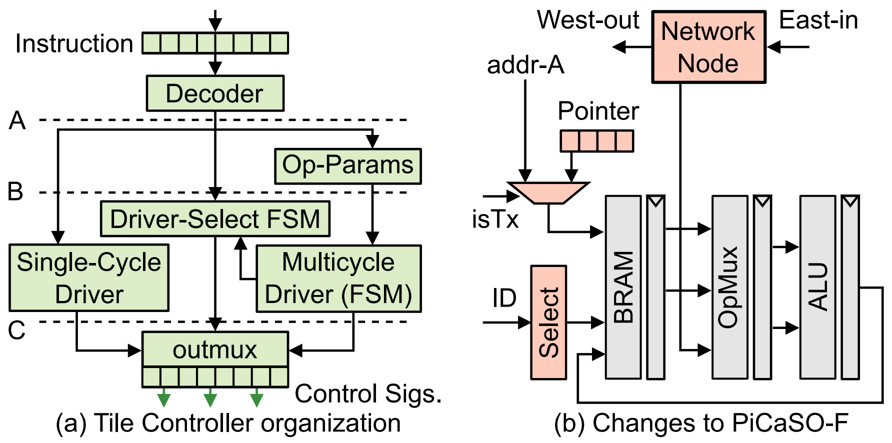

Figure 2: Architectures of (a) GEMV controller and (b) PIM block (PiCaSO-IM).

2 Built-in Instructions

These are instructions built into IMAGine. Each instruction mnemonic generates exactly one instruction word. However, some instruction may be executed by the hardware in multiple cycles. Instructions prefixed with `mv_` are dispatched to the GEMV array submodule and the instructions prefixed with `vv_` are dispatched to the column-shift-register submodule. Submodules can concurrently execute their instructions unless a synchronization barrier is used.

The register arguments (e.g. rs1, rs2, rd, reg, etc.) are integers greater than 0. The largest value of the integer arguments depends on the assembler parameters and the instruction. The optional keyword argument `comment` after * can be used to emit user comments in the outputs generated by the export directives. Other keyword arguments (e.g. skipChekcs) after * can be safely ignored for programming purposes.

mv_write (self, addr, data, *, comment=None)

This is a single-cycle instruction for the GEMV array. Writes the `data` at the `addr` of the BRAM (PiCaSO) block(s) currently selected. Use one of the select instructions to select the BRAM before write: `mv_selectBlk`, `mv_selectRow`, `mv_selectCol`, and `mv_selectAll`.

mv_nop (self, *, comment=None)

This is a single-cycle instruction that generates a NOP for the GEMV array. This effectively consumes one cycle without changing internal state.

mv_mov (self, rd, rs, *, comment=None)

This is a multicycle instruction for the GEMV array. It moves (copies) the contents of the `rs` register into the `rd` register.

mv_add (self, rd, rs1, rs2, *, comment=None)

This is a multicycle instruction for the GEMV array. It adds the contents of the registers `rs1` and `rs2` and stores the resutl in the register `rd`. This is effectively $rd = rs1 + rs2$.

mv_sub (self, rd, rs1, rs2, *, comment=None)

This is a multicycle instruction for the GEMV array. It the contents of `rs2` from `rs1` and stores the resutl in the register `rd`. This is effectively $rd = rs1 - rs2$.

mv_movOffset (self, offset, rd, rs, *, comment=None, skipChecks=False)

This is a multicycle instruction for the GEMV array. It moves (copies) the contents of the `rs` register, starting at bit position `offset`, into the `rd` register. This instruction does not respect the register boundary. This means, the upper bits of `rd` will contain `offset` number of lower bits of the `rs+1`-th register. This instruction is used after multiplication to perform right-shift of the multiplication result stored in two consecutive registers.

mv_selectBlk (self, rowID, colID, *, comment=None)

This is a single-cycle instruction to select a single PiCaSO block for selective operations like `mv_write`. To select a specific block both `rowID` and `colID` are needed.

mv_selectRow (self, rowID, *, comment=None)

This is a single-cycle instruction to select all PiCaSO blocks in an entire row of the GEMV array for selective operations like `mv_write`. Only `rowID` is needed for the row selection.

mv_selectCol (self, colID, *, comment=None)

This is a single-cycle instruction to select all PiCaSO blocks in an entire column of the GEMV array for selective operations like `mv_write`. Only `rowID` is needed for the row selection.

mv_selectAll (self, *, comment=None)

This is a single-cycle instruction to select all PiCaSO blocks in the GEMV array for selective operation like `mv_write`.

mv_updatepp (self, ppreg, multiplicand, multiplier, bitNo, *, comment=None)

This is a multicycle instruction to update the partial-product based on the `bitNo`-th bit of the multiplier. The result is equivalent of,

```
ppreg[bitNo +:  N] += multiplicand * multiplier[bitNo]; // N = register width
ppreg[bitNo + N] = ppreg[bitNo + N - 1]; // Sign extension
```

mv_blockFold (self, fold, rd, rs, *, comment=None)

This is a multicycle instruction to add a fold of `rs` with itself and store the result in the destination register `rd`. Here, `rd` and `rs` can be the same register. The result is equivalent of,

```
rd = rs + folded(rs, fold).
```

mv_accumRow (self, level, reg, *, comment=None)

This is a multicycle instruction to add the block-level accumulation result of different blocks in a row based on the accumulation tree `level`. The result is equivalent of,

```
receiver-pe0-reg += transmitter-pe0-reg
```

vv_nop (self, *, comment=None)

This is a single-cycle instruction that generates a NOP for the column-shift-register submodule. This effectively consumes one cycle without changing internal state.

vv_shiftOff (self, *, comment=None)

This is a single-cycle instruction that turns off all types of shifting in the column-shift-register submodule.

vv_serialEn (self, *, comment=None)

This is a single-cycle instruction that enables serial shifting from GEMV array into the column-shift-register submodule. This automatically disables parallel shifting.

vv_parallelEn (self, *, comment=None)

This is a multicycle instruction that enables parallel shifting from GEMV array into the column-shift-register submodule. This automatically disables serial shifting.

3 Assembler Macros

These are high-level macro-instructions provided by the assembler, built on top of the built-in instructions. Each instruction mnemonic exapands into several (sometimes hundreds) built-in instruction words. The programmers are encouraged to use the macros as much as possible to avoid the intricacies of the built-in instructions.

mv_MULT (self, rd, multiplicand, multiplier, *, comment=None, skipChecks=False)

This instruction performs signed multiplication between the registers `multiplicand` and `multiplier` using the `mv_updatepp` built-in instruction. The result is stored spanning two registers `{rd, rd+1}`.

mv_MULTFXP (self, rd, multiplicand, multiplier, *, comment=None)

This instruction performs signed fixed-point multiplication between the registers `multiplicand` and `multiplier` using the `mv_updatepp` and `mv_movOffset` built-in instructions. This instruction depends on the assembler parameter `fracWidth` and reserved registers.

mv_SYNC (self, *, comment=None)

This is a synchronization barrier for the GEMV array. Next instruction will not be fetched and executed until the last `mv_` instruction finishes. This instruction can be used to stall the execution of an instruction on the column-shift-register submodule, which needs the result of the last instruction dispatched to the GEMV array submodule.

vv_SYNC (self, *, comment=None)

This is a synchronization barrier for the column-shift-register submodule. Next instruction will not be fetched and executed until the last `vv_` instruction finishes. This instruction can be used to stall the execution of an instruction on the GEMV array submodule, which needs the last instruction dispatched to the column-shift-register to finish first.

mv_BLOCKACCUM (self, rd, rs, *, comment=None)

This instruction performs the block-level accumulation of the `rs` register and saves the result in the `rd` register using the `mv_blockFold` built-in instruction.

mv_RNGACCUM (self, colCnt, rd, rs, *, comment=None)

This instruction accumulates the `rs` register of PE columns (0 : colCnt-1) into `rd` register using a combination of `mv_blockFold` and `mv_accumRow`.

mv_ALLACCUM (self, rd, rs, *, comment=None)

This instruction accumulates the `rs` register of all PE columns in a row into `rd` register using a combination of `mv_blockFold` and `mv_accumRow`.

mv_CLRREG (self, reg, *, comment=None)

This instruction clears the register `reg` by writing zeros to the corresponding rows of the BRAM block.

mv_LOADMAT (self, reg, matrix, *, comment=None)

This instruction generates instruction to load a floating-point (or integer) matrix into the register `reg` of the GEMV array.

It performs the necessary bit-manipulations to convert the floating-point (or integer) values to transposed bit-patterns for the corresponding fixed-point representation. It is compatible with numpy, i.e, the `matrix` can be a numpy ndarray instance. This macro is data aware: it generates the instructions to clear the register first and then write only the non-zero rows to the target BRAMs. Thus, if a matrix with all zeros is loaded, it only clears the register.

mv_LOADVEC_ROW (self, reg, vector, *, comment=None)

This instruction generates instruction to load a floating-point (or integer) row-vector into the register `reg` of the GEMV array. The same vector is loaded into all rows of the GEMV array.

It performs the necessary bit-manipulations to convert the floating-point (or integer) values to transposed bit-patterns for the corresponding fixed-point representation. It is compatible with numpy, i.e, the `vector` can be a numpy ndarray instance. This macro is data aware: it generates the instructions to clear the register first and then write only the non-zero rows to the target BRAMs. It takes advantage of the broadcast network connected to the GEMV array to write to all rows at the same time. Thus, it is a better choice than `mv_LOADMAT` if a matrix needs to be loaded which has identical rows.

mv_LOADVEC_COL (self, reg, vector, *, comment=None)

This instruction generates instruction to load a floating-point (or integer) column-vector into the register `reg` of the GEMV array. The same vector is loaded into all columns of the GEMV array.

It performs the necessary bit-manipulations to convert the floating-point (or integer) values to transposed bit-patterns for the corresponding fixed-point representation. It is compatible with numpy, i.e, the `vector` can be a numpy ndarray instance. This macro is data aware: it generates the instructions to clear the register first and then write only the non-zero rows to the target BRAMs. It takes advantage of the broadcast network connected to the GEMV array to write to all columns at the same time. Thus, it is a better choice than `mv_LOADMAT` if a matrix needs to be loaded which has identical columns.

4 Assembler Directives

These are assembler directives to control various aspects of the assembly and instruction generation process.

setupParams (self, regCnt, regWidth, maxLevel, maxFold, idWidth, fracWidth, mvBlockDim, resvRegCnt)

This directive sets up the assembler parameters. The assembly programmers should avoid using this function directly and use the `loadParams()` directive instead to load the appropriate parameters. Default values of the parameters are,

regCnt = 16, regWidth = 16, maxLevel = 3, maxFold = 4,
idWidth = 8, fracWidth = 0, mvBlockDim = None, resvRegCnt = 0

loadParams (self, filepath, cpright=True, showparams=True)

This directive loads the assembler parameters from and external YAML file.

reset (self)

This directive resets the internal state of the assembler and clears the instruction cache, without affecting the assembler parameters. This directive should be called before starting a new program and after exporting the previous program in a multi-program assembler script.

assemble (self, verbose=False)

This directive assembles the instructions in the instruction cache and convert them into their machine-code representation. It does not generate any output, only updates the internal state of the assembler. It is usually not necessary to be called directly by the programmer. Any of the export directives will implicitly call it if there is even a single unassembled instruction in the cache.

export_verilogBin (self, filename=None, comment=True, source=True, separator='_')

This directive exports the assembled instructions as binary string with the `separator` between the instruction word fields. The exported output can be used in Verilog/SystemVerilog simulation tools using the $readmemb() system task.

export_CprogHex (self, progname, filename=None, comment=True, source=True)

This directive exports the assembled instructions as unsigned hex number C-array. The exported output can be compiled into object code using a C compiler to be used in the application program, along with necessary header files.

export_CprogHeader (self, filename=None)

This directive exports a C-header file containing the definition of the struct that can hold all the necessary information and the compiled instructions.

as_addComment (self, comment)

This directive looks like an instruction and can be used to emit comments in the outputs generated by the export directives.