Jetson TX2 - Processing Components

- Dual-core NVIDIA Denver2 + quad-core ARM Cortex-A57
- 256-core Pascal GPU
- 8GB LPDDR4, 128-bit interface
- 32GB eMMC
- 4kp60 H.264/H.265 encoder and decoder
- Dual ISPs (Image Signal Processors)
- 1.4 Gpps MIPI CSI camera ingest
Jetson TX2 - Denver2 and ARM

- IMPORTANT: we do not use Denver2 cores in our assignments. This is just for explaining the architecture
- Let’s check the CPUs:
Jetson TX2 - Denver2 and ARM

- IMPORTANT: we do not use Denver2 cores in our assignments. This is just for explaining the architecture.
- Let’s check the CPUs.
- Query current CPU configuration:
  ```bash
  sudo nvpmodel -q
  ```
- Check available CPU configurations:
  ```bash
  cat /etc/nvpmodel.conf
  ```
- Set current CPU configuration:
  ```bash
  sudo nvpmodel -m <id>
  ```
SISD, SIMD, MIMD, & MISD

- SISD: Single Instruction, Single Data
  one processor that handles one algorithm using one source of data at a time
SISD, SIMD, MIMD, & MISD

- **SISD**: Single Instruction, Single Data
  one processor that handles one algorithm using one source of data at a time

- **SIMD**: Single Instruction, Multiple Data
  several processors that follow the same set of instructions, but each processor inputs different data into those instructions
SISD, SIMD, MIMD, & MISD

- **SISD:** Single Instruction, Single Data
  one processor that handles one algorithm using one source of data at a time

- **SIMD:** Single Instruction, Multiple Data
  several processors that follow the same set of instructions, but each processor inputs different data into those instructions

- **MIMD:** Multiple Instructions, Multiple Data
  multiple processors, each capable of accepting its own instruction stream independently from the others. Each processor also pulls data from a separate data stream
SISD, SIMD, MIMD, & MISD

- **SISD: Single Instruction, Single Data**
  One processor that handles one algorithm using one source of data at a time

- **SIMD: Single Instruction, Multiple Data**
  Several processors that follow the same set of instructions, but each processor inputs different data into those instructions

- **MIMD: Multiple Instructions, Multiple Data**
  Multiple processors, each capable of accepting its own instruction stream independently from the others. Each processor also pulls data from a separate data stream

- **MISD: Multiple Instructions, Single Data**
  Multiple processors. Each processor uses a different algorithm but uses the same shared input data
SISD, SIMD, MIMD, & MISD

- **SISD: Single Instruction, Single Data**
  one processor that handles one algorithm using one source of data at a time

- **SIMD: Single Instruction, Multiple Data**
  several processors that follow the same set of instructions, but each processor inputs different data into those instructions

- **MIMD: Multiple Instructions, Multiple Data**
  multiple processors, each capable of accepting its own instruction stream independently from the others. Each processor also pulls data from a separate data stream

- **MISD: Multiple Instructions, Single Data**
  multiple processors. Each processor uses a different algorithm but uses the same shared input data
SISD on ARM

- **SISD**: Single Instruction, Single Data
  one processor that handles one algorithm using one source of data at a time
- Example: your previous assignment ran sequentially on CPU cores
SIMD on ARM

- SIMD: Single Instruction, Multiple Data
  several processors that follow the same set of instructions, but each processor inputs different data into those instructions
SIMD on ARM

- SIMD: Single Instruction, Multiple Data
  several processors that follow the same set of instructions, but each processor inputs different data into those instructions
ARM Neon Programming

- ARMv8 Neon Unit:
  - Fully integrated into the processor and uses processor’s resources for loop control, caching, and integer operations
  - Uses 128-bit registers for SIMD processing
  - It’s register file is a collection of registers that can be accessed as (8, 16, 32, 64, 128)-bit registers
  - Registers contain vector of elements. The same element position in the input and output registers is referred to as a lane
  - Each Neon instruction results in “n” parallel operation, where “n” is the number of lanes
Neon Register and Element Size

- 128-bit register
  - 2 x 64-bit elements: 2D (Double word)
  - 4 x 32-bit elements: 4S (Single word)
  - 8 x 16-bit elements: 8H (Halfword)
  - 16 x 8-bit elements: 16B (Byte)
Neon Register and Element Size

ARMv8 (Jetson)
32 x 128-bit vector registers
31 x 64-bit general purpose registers

ARMv7 (PYNQ)
16 x 128-bit vector registers

Neon Intrinsics

- Are functions calls that compiler replaces with an (or a sequence of) appropriate Neon instruction(s)
Neon Intrinsics

- Are functions calls that compiler replaces with an (or a sequence of) appropriate Neon instruction(s)

Data types:

- D
- S
- H
- 8B (Byte)
- 4H (Halfword)
- 2D (Double word)
- 4S (Single word)
- 8H (Halfword)
- 16B (Byte)
- 1D (Double word)
- 4S (Single word)
- 4H (Halfword)
- 8B (Byte)
Neon Intrinsics

- Are functions calls that compiler replaces with an (or a sequence of) appropriate Neon instruction(s)

- Data types:
  \(<\text{int, uint, float, poly}>64, 32, 16, 8\times16, 8, 4, 2, 1>_t\)
Compile Neon

- ARMv7 (PYNQ) requires `-mfpu=neon` and `-O1 -ftree-vectorize`
- ARMv8 (Jetson) requires `-O1 -ftree-vectorize`
- **Lab Work:**
  - Complete `neon.c` (provided)
  - Compile it with: `gcc -mfpu=neon neon.c -o neon`
  - Run: `./neon`
Compile Neon

- ARMv7 (PYNQ) requires 
  -mfpu=neon and -O1 -ftree-vectorize
- ARMv8 (Jetson) requires -O1 -ftree-vectorize

Lab Work:
- Complete neon.c (provided)
- Compile it with: gcc -mfpu=neon neon.c -o neon
- Run: ./neon
- Modify the code to add 250 to data instead of 3
FIR Filtering
1D Convolution
1D Convolution
1D Convolution
1D Convolution
1D Convolution

- 2 nested loops
- Loop through (size of input - size of filter)
  - For each filter coefficient
    - Multiply by the input and accumulate
  - Store result in the output
1D Convolution

- Complete the naive implementation in src/fir.cpp
Loop Unrolling

Unroll coefficient loop (inner loop) by 4:

- Manually duplicate the single line of code
- Increment loop variable by 4
Gprof Profiling

- Profiling tool (like perf), but will provide information on independent function calls within the executable
  - Perf will only provide a cycle count and execution time for the entire executable.
- Compile flag `-pg`
- Running the executable
- There should now be a report `gmon.out` in the directory
- **Make sure you remove the gmon.out if you run the program again.**
Gprof Profiling

- View the report with `gprof -b <EXEC-NAME> gmon.out`
  - For this lab, the command is `gprof -b lab3_fir gmon.out`
- `fir()` takes up 50% of execution time
- `fir_opt()` takes up 42.9% of execution time
SIMD Instructions

- Include `<arm_neon.h>` (already done for the lab)
- Add compiler flag: `-mfpu=neon` (only on PYNQ, not on Jetson. Already done for lab)

Replace the unrolled loop body by NEON Instructions

1. Declare SIMD registers: Use 128-bits SIMD vectors
   a. Float 32-bit x 4
2. Initialize output SIMD vector with 0
3. Inside the loop:
   a. Load input data into SIMD vector
   b. Load coefficients into SIMD vector
   c. Multiply-accumulate into output SIMD vector
4. Add 4 values together then store in output array
Compile Comparison

- Compile the lab with the -O0 compilation flag
- Run the executable and investigate the gprof report `gprof -b lab3_fir gmon.out`
Compile Comparison

- Change the compile flag from -O0 to -O1
- Run the executable and investigate the gprof report `gprof -b lab3_fir gmon.out`

https://linux.die.net/man/1/gcc