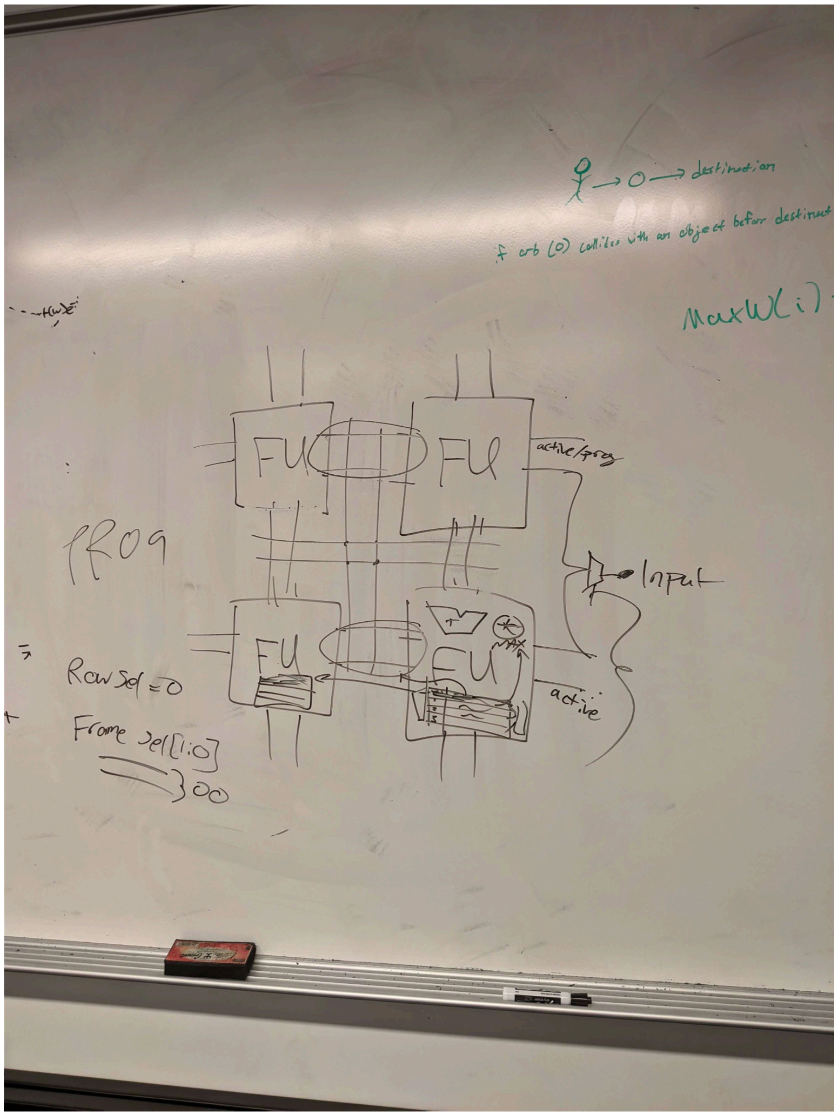
Background & Motivation

We chose to do this project mainly because it's something unique that hasn't been taped out before over the course of this class. Our project is a coarse grained reconfigurable accelerator (CGRA), which is in essence a higher level version of an FPGA. This is unique as most other designs that have successfully reached tapeout in past semesters have been derived from RISC-V CPUs, so attempting something quite different poses its own set of challenges for us to resolve. A CGRA is also very versatile as it (as the name suggests) can be reconfigured to accelerate various different applications including cryptography, neural networks, graph computation, and more. Our design also accounts for using multiple chips together, as we plan to allow them to be daisy-chained to basically form a systolic array, as well as be able to perform computations in parallel over large data sets. The project itself is also scalable in terms of difficulty and size, as we can add more functional units, increase/decrease the complexity of said functional units, and overall make the design as large or as small as we want. This is important as we plan to pursue taping out on the Intel 16nm process if possible, so if we are able to complete the scripts for that process in time we can properly utilize the increased area and pinout by easily scaling our design up.

Design Plan & High-level Architecture



At a high level, our architecture will consist of a connected mesh of functional units (FUs). As of right now, we are considering these to all be the same but consist of various compute elements within. This is subject to change as we better understand our use cases, in which case we may use different categories of functional units. However, we plan on designing this with a frame-based programming strategy to minimize reconfiguration time. In essence, a user will be able to program some limited number (4) of “frames” dictating functional unit uses (multiply, add, etc.) and mesh connections.

We anticipate programming to be done serially with a combination of bitstreams to provide frame data and row selection. The general strategy is to select a row of FUs to program and which frame to write to. Then, a one-bit input stream is connected as an input to the first FU in the selected row. This data will then be shifted through the row (basically just a large shift register made up of each FU’s frame registers concatenated). We hope that this simpler programming design will be made up for by the ability to pre-program frames, though we are aware that this will cause programming to take up significantly more time at the start of a program.

While we have not finalized what the connection architecture will look like, the “node” connections around a given FU will be specified by its frame data. This will allow us to not only reconfigure FU execution on-the-fly, but also how they are interconnected within the chip by swapping the in-use frame. For example, one frame may implement a multiply-accumulate algorithm involving data being sent to multipliers, then accumulated in stages. A second frame could simply be element-wise vector addition. These could be swapped between by simply adjusting the frame select lines accordingly.

One of the most difficult considerations for this project is what the dataflow into and out of the chip will look like. As of right now, we are considering a serialized dataflow, allowing for wider internal bus width despite a smaller external bus. Given that most use cases will involve element-wise operations, we can use 16 pins as serialized data input and 16 as data output. Of course, this pincount can scale tremendously with Intel’s process, allowing us to potentially send upwards of 64 serialized elements at time. Then, we will use some sort of data size selector (which could be programmed in with the frame data) to determine how many cycles to wait for data to fully process. One consideration we still have to make is whether or not data will be sent directly from a host CPU or if we will have a small core within the chip that can manage talking with external memory. If data is to be sent directly, there will likely be small register files within each FU to store its data.

An additional design consideration is how data will flow between FUs within the chip for a multi-stage frame, as well as how the chip will externally signal that data is ready. This is not currently finalized, but we plan on referencing SambaNova’s SN40L as inspiration for our design. Lastly, we also may need to consider a better form of on-chip or per-FU storage than a register file due to area concerns, such as possibly using an SRAM cache, which then results in a few more considerations in terms of data streaming for I/O.

In terms of post-fabrication testing, we intend on designing a PCB that is populated with a host CPU (some microprocessor like an STM32) and potentially several of our chips in parallel. These will be written to with memory-mapped I/O to make serializing data easier. As we design our CGRA, we will be considering potential use cases such as ML training, cryptography acceleration, or graph algorithms. These will provide the test scenarios to run on the microprocessor using our chip as an accelerator. The most difficult part of testing will be correctly synthesizing and communicating bitstreams for programming the CGRA, which will involve designing a low level format that describes our frames (and all other reconfigurable options), as well as a software system to program the CGRA accordingly.

We plan on developing a compiler that targets our architecture for code generation. This will allow for advanced benchmarks to be run on our design, and show how our design would integrate as part of a larger system. We are considering supporting opsets such as TinyGrad, or simple cryptographic operations, as initial frontends. We eventually may integrate with a more sophisticated and pre-existing compiler infrastructure, to leverage pre-existing work and optimizations.

The nature of our architecture is well suited for compiler optimizations. Two key observations are our limitation of control flow (dataflow architecture), and opportunity for reprogrammability. Due to the static nature of the computations we will support, we can derive data access patterns at compile time. This enables lots of interesting ideas such as a host managed or pre-programmed cache on the CGRA itself, allowing for input data to be efficiently accessed. Another consequence is that there will not be much need for dynamic scheduling of operations in the hardware itself. Instead this can be done at compile time and in a target specific way, so it can be optimal. Finding optimal mappings and scheduling of frames for a program across available functional units is a difficult problem that we will attempt to solve via vibes and heuristics.

Timeline and Milestones

- Summer:
    - Finish architectural design (All)
    - Make RTL design (Primarily Ethan, Udit, Jason)
    - Make software model (Primarily Ryan, Xavier)
    - Verify RTL functionality against software model (All)
    - Figure out Intel 16nm params/scripts? (Primarily Ryan at the start of summer, we will see how it progresses)
- During semester:
    - Split efforts along getting TSMC to work vs figuring out Intel to guarantee we get a baseline chip ready for tapeout (Ryan and someone else on Intel, other two on TSMC)
        - Will develop a more in-depth timeline soon
    - Work on the compiler backend necessary for proper functionality (Primarily Xavier)

Main Challenges and Complexities

- Strengths:
    - Easy to scale up or down depending on the end process
    - Easy to reconfigure for various use cases
- Weaknesses:
    - Limited bandwidth on TSMC process (too few pins)
    - Requires significant work on the compiler backend

    - May take a relatively long time to reconfigure (based on current design)
- Difficulties:
    - Communication with host CPU (mem-mapped I/O may require CPU modifications?)
    - Managing large datasets efficiently and quickly
    - Designing the reconfigurable interconnects

Required IPs *(if any)*

We will need IPs for serialization and deserialization, SRAM of some sort, and depending on functional units we may need more but we are not sure at the moment. This could include IPs for multipliers, dividers, etc. to make the RTL design focus higher level and centered more around the interconnects between the functional units rather than the functional unit microarchitecture.

Member Backgrounds

- Ethan
    - Undergrad senior in fall
    - Taken ECE 425, ECE 411, RTL design internship
    - Staying for bringup
- Jason
    - Undergrad senior in fall
    - Taken ECE 425, ECE 411
    - Staying for bringup (as masters, hopefully)
- Ryan
    - Graduate student in fall
    - Taken ECE 425, experience with compiler development from personal projects
    - Staying for bringup
- Udit
    - Undergrad senior in fall
    - Taken ECE 425, ECE 411, ECE 482, CPU design internship
    - Staying for bringup maybe (depends on summer)
- Xavier
    - Undergrad senior in fall
    - Taken ECE 411, CS 426, 526, compilers research + compilers internship
    - Staying for bringup