Hello everyone. Thanks for coming to my presentation, and I'm happy to have the opportunity to present our work: ActiveN, A scalable and flexibly-programmable event-driven neuromorphic processor.

First I'm going to talk a little about the motivation of our work. The energy efficiency of neuromorphic computation comes from its event-driven nature, where computations can be saved when there is nothing happening. This does mean that the computation itself presents sparsity in both the spatial and temporal sense. Now this sparsity in and of itself is not a problem. Previous works on neuromorphic architectures tolerate, or even benifits from this sparsity, by using dataflow architectures coupled with custom circuitry to do event-driven computation.

However we do notice a few trends in recent studies. The first is that we're focusing on more diverse neuron models, both for the neuron update procedure itself, and for the postsynaptic weight accumulation. This requires at least some sort of programmability in our architecture if we want to reuse an processor for different models. The conflict is that contemporary general-purpose processor architectures, such as CPUs and GPGPUs, cannot efficiently deals with this sparsity. Instead, this sparsity breaks locality, causes synchronization, and so on. So we lose the energy efficiency, and we also lose the performance.

The second trend is that the size of the models are getting larger. For example, people ar already running models > 4M neurons on GPU and investigating its dynamics. The problem is that high density memories such as DRAMs also really hates this sparsity. So we also lose the efficiency and performance here, especially these larger memories have higher latencies.

So the question we ask is that basically we are trying to do it from the other way around: Is it possible to draw inspirations from dataflow neuromorphic architectures, and introduce them as architectural extensions onto a general purpose one, and get a efficient programmable neuromorphic processor? Our answer is yes, but with an asterisk, so we do have to make some small restrictions on the SNN model, which we will talk about more in the following parts of this presentation.

Now we're going to share some of the core insights we gained through our analysis.

We start by characterizing the memory accesses involved in an SNN simulation. Assume a general parallel architecture with a high density global memory, and a neuron currently being processed on a PU fires. So the update itself would access its state, and when it fires, the spike delivery procedure need to acquire its neighboors by reading the synapse data, buffer the spikes somewhere if needed, and finally modify neuron inputs. We can figure out the size, lifetime, and accessing behavior of these types of data. Now the goal is to make most memory accesses continous, or at least those targeted at long latency global memory.

We do this by first, make synapse data access dense. This is easy, since all accesses to the synapse matrix is in a per-row basis, so if the matrix is stored in compressed sparse row format, the access is essentially continous. Besides that, we want to eliminate spike data storage completely, and because access to neuron inputs are inheriently sparse, we also want to introduce a sratchpad memory to each PU to store neuron states and inputs, which provides single-cycle latency even for sparse accesses. For the second point, how can spike be timely consumed would be a problem, and for the last point, neuron states are only accessed locally assuming static mapping of models onto PUs, but inputs needs to be accessed remotely.

We solve these problems by a unified architectural features: active messaging. We gives PU the ability to send messages with data to other PUs, which are handled by event-triggered handlers. Now instead of writing spike into a global buffer and synchronize explicitly, we can send the spike as messages to its destination, which also serves as a synchronization primitive. When a PU receives

a spike, it updates its own private memory. We then can let the priority of incoming spikes be higher than neuron updates for its timely consumption.

Now that we dealt with the sparsity, and the synapse matrix is placed in CSR format in a global high density memory, the second thing we want deal with is the latency of accessing the synapse matrix. A simple optimization would be putting CSR row pointers into scratchpads, which can save us one roundtrip to global memory. However, since reads to synapse data cannot be eliminated, and its size requires it to reside in global memory, we need other way to tolerate at least one roundtrip latency. The traditional way is to either block (typical for a in order processor), or to introduce numerous MSHRs and additional registers to hold the states.

Instead we exploit another feature of the SNN simulation: The read of the synapse matrix is in some sense "context-free". If we explicitly expand the load operation as a asynchronous operation, as shown in the pseudo-code here, we can notice that the operations after the load value returns doesn't depend on the state from before the load is executed. So if we drop all the previous states, the spike propagation procedure can still correctly continues. This mean that we can have some sort of a "fire-and-forget" style asynchronous memory accesses, where a PU send a memory access to the memory, and then continue to do other unrelated works, which is free to destroy any architectural states. When load values are returned, it's processed in a brand new context. This may also seems very similar to an "event-driven" style of doing memory accesses.

So that's exactly how we implement it, by sending memory requests based on active messaging. We reserved message types exclusively for asynchronous loads, asynchronous stores and asynchronous memory responses. Now that PUs can do other work when memory accesses is inflight, for examples, handling incoming spikes or updating neurons. This way core doesn't block for inflight memory accesses, but anso doesn't need context tracking and parallelism structures: MSHRs, warp queues, SMT threads, additional physical registers, etc. This also means number of concurrent memory accesses no longer restricted by these architectural constraints, which can lead to much higher achievable memory bandwidth.

As a side note, this style of asynchronous accesses can also be applied to memory accesses that want to keep some context. Software can manually saves the registers into its scratchpad. This might still be benificial comparing against introducing a separate hardware thread if the amount of saved state is relatively small, for example, one to two registers, because scratchpad access is guaranteed to be fast.

Finally, we introduced a simple but very effectively optimization for those truly context-free memory access to a synapse matrix. Notice that when the data returns to the PU, the PU simply forwards it's weight to postsynaptics PU. Because the processing also does not depend on any context (or state) on the presynaptic PU, it's possible to let the memory controller interprets the CSR format inside loaded data and directly send it to postsynaptic PU, saving one forward, which basically halves NoC traffic.

So this require us to allocate another message type, we call that readCSR, which sends the base address and the length of the read. The memory controller reads the data, figure out which destination this spike is sent, and encapsulate a message directly to the destination. To sum up, this is the overview of how SNNs execute on top of ActiveN: PUs hold neuron states, CSR row offsets and update queue inside it's private scratchpad. When a neuron fires, presynaptic PUs send messages to memory controllers to load synapse matrix, and the memory directly encapsulate the loaded data as outgoing spikes and send to postsynaptic PU.