Emulating a Retrieval Accelerator

You are tasked with building a simplified software emulation of our retrieval accelerator that retrieves documents encoded into a sparse, binary-weight bag-of-words (BoW) representation. This encoding scheme defines the input interface of our accelerator: a BoW model with a vocabulary of size $|V|$ defines an accelerator with $|V|$ input wires, and when a document d (or query) is encoded into a sparse bag of words from that vocabulary, $E(d) = \{v_i | v_i \in V \wedge v_i \in d\}$, that defines an equivalent set of active/high-voltage wires $\{i | i \leq |V| \wedge v_i \in E(d)\}$.

This accelerator operates in three phases: (1) **memorization**, during which each document's BoW vector is sequentially programmed into the chip's memory, (2) **search**, during which a query's BoW vector is presented to the chip to score the memories, and (3) **ranking**, during which the highest scoring memories stream out of the accelerator in decreasing relevance. To perform these operations, the accelerator consists of three modules: a **scoring unit** for each document in the corpus, a programmable **routing network** that connects a subset of the $|V|$ input wires onto each scoring unit, and a **ranking network** that collects and compares the output of scoring units and streams out their index in order of decreasing score.

Your main task will be to architect and implement a framework to model these modules and operations. In addition, your design must include an evaluation script that benchmarks the accelerator's performance with Recall@1000.

Document Retrieval Dataset

We've provided you with a small collection of documents $\mathcal{D}$, queries $\mathcal{Q}$, and ground-truth relevance labels taken as a subset of MS MARCO. The documents can be found in `mini_msmarco/full_collection/raw.tsv`, organized as `<pid>:tab:<passage_text>:newline:`. The queries can be found in `mini_msmsarco/dev_queries/raw.tsv`, organized as `<qid>:tab:<query_text>:newline:`. The gold-truth relevance labels are given in `mini_msmarco/dev_qrel.json`, a dictionary where the key is a `qid` and its value is a dictionary where each key is the `pid` of a relevant passage.

Bag-of-Words Encoder

This module is not implemented within the accelerator, but must be designed to interface between text and the accelerator's input wires. Feel free to choose whichever vocabulary V you are familiar with to tokenize the text. A document d is encoded into a set $E(d) = \{v_i | v_i \in V \wedge v_i \in d\}$ and a query q is encoded into a set $E(q) = \{v_i | v_i \in V \wedge v_i \in q\}$.

Memorizing Documents in a Programmable Routing Network

A routing network is designed to connect some number of input wires ($|V|$) to some number of scoring units ($|\mathcal{D}|$). Memorization is the act of configuring those connections for each (scoring unit, `pid`) pair. In general, a router may need to provide each scoring unit with parallel access to multiple input wires. However, for the purposes of this task, you are tasked with implementing the simplest routing network known as a crossbar network, with $|V|$ input wires and $|\mathcal{D}|$ output wires. A crossbar network is a collection of switches arranged

in a matrix configuration that allows any input wire to connect to any output wire. This network does not provide parallel access of multiple input wires to each output wire and scoring unit, so inputs to the crossbar must be presented serially.

Searching for Documents with Scoring Units

A scoring unit is designed to receive inputs from the routing network and yield a numeric score. Inputs may arrive in parallel, serially, or both. The scoring unit continuously updates its numeric score as inputs arrive. For a scoring unit to correctly measure document relevance with a BoW encoding, as this task requires, implement a specific type of scoring unit that simply counts the number of active signals it receives from the crossbar network, incrementing this count with every additional signal.

Retrieving Documents with a Ranking Network

When triggered, the ranking network begins streaming the index of different scoring units in order of decreasing score, up to a maximum of R steps. Implement this module as a Maximum Binary tree that fills a binary tree with scoring units such that the root node is a scoring unit with the highest numeric score. Feel free to break ties arbitrarily. After filling the tree, it should yield a length-R stream of scoring unit identifiers by removing the top node and updating the tree R times.

Benchmarking Recall

Write a script to evaluate the retrieval performance of this accelerator. It should first use the BoW encoder to encode documents and then present and memorize them sequentially in the routing network. This is the **indexing** phase of the process. Afterwards, it enters the **retrieval** phase and encodes a query, presents it to the router which updates scores in each scoring unit, and then triggers ranking and receives a length-1000 stream of identifiers for the top-1000 scoring units. Run retrieval for all queries in the provided collection and log the results to a file. In addition, compute Recall@1000 using the provided ground-truth relevance labels.

Going further

If you'd like to extend this further, below are a few features you could implement.

- How much energy does the routing network consume per query? Energy-use comes from sending signals along wires, heating up the chip, and is proportional to wire-length. Consider writing a script to measure energy consumption.

- How could we modify the scoring units to support an integer-valued BoW encoder that counts the number of term occurrences in a document and query? Consider implementing an alternative encoder module and scoring unit module and evaluate its Recall@1000.

- How many stages are needed in the binary-tree based ranking network, and how does that affect latency? Consider implementing a Max-Tree network with higher degree to achieve lower retrieval latency.