Accelerating RAG Using AI Engines on the Xilinx Versal VCK5000: A Study of Dense Retrieval, GEMM Optimizations, and Multi-Tile Scaling

Sai Divya Ravipati[1*], Damini Chandi Priya A[1*], RadheShyam M[1*]
International Institute of Information Technology Hyderabad (IIITH)[1]
*These authors contributed equally
Email: {sai.divya, radheshyam.modampuri}@students.iiit.ac.in, daminichandipriya.a@research.iiit.ac.in

***Abstract*—This paper presents an accelerated dense-retrieval architecture for Retrieval-Augmented Generation (RAG), targeting high efficiency similarity search on the Xilinx Versal VCK5000 AI Engine platform. Dense semantic retrieval is a core component of Retrieval-Augmented Generation (RAG) systems, yet its performance is fundamentally limited by the memory bound nature of dot-product similarity computation between queries and large embedding databases. CPUs and GPUs repeatedly stream the full database for each query, resulting in low operational intensity and underutilized compute resources. This work aims to accelerate exact dense retrieval on the Xilinx Versal VCK5000 AI Engine platform by reformulating retrieval as optimized matrix–vector and matrix–matrix multiplication. We partition the embedding database across multiple AI Engine tiles and evaluate two dataflow strategies: an inner-product approach using packet-streamed database tiles with broadcasted queries, and an outer-product approach that streams k-blocks once and performs tile-local accumulation. Analytical modeling and empirical measurements show that while the inner-product baseline remains memory-limited, the outer-product formulation substantially improves data reuse, increases operational intensity, and shifts execution toward compute-bound behavior. Our implementation achieves 8.06 GFLOP/s on the VCK5000 far exceeding CPU performance and approaching GPU throughput demonstrating the effectiveness of algorithm architecture co-design for scalable, low-latency RAG retrieval.**

***Index Terms*—Retrieval-Augmented Generation (RAG), dense retrieval acceleration, batched similarity search, vector database partitioning, AI Engine architecture, Versal VCK5000, matrix–vector multiplication, matrix–matrix multiplication, tiled compute arrays, operational intensity.**

I. Introduction

Low-latency and knowledge-grounded inference are increasingly required in modern AI applications such as question answering, enterprise search, recommendation engines, and real-time decision support. In these scenarios, Large Language Model (LLM) systems must dynamically access external information sources instead of relying solely on parametric knowledge. Retrieval-Augmented Generation (RAG) has therefore emerged as a crucial architectural framework, enabling generative models to query external vector databases or document repositories and incorporate retrieved context into generated responses. By grounding responses in external evidence, RAG significantly reduces hallucination and improves factual accuracy and reliability.

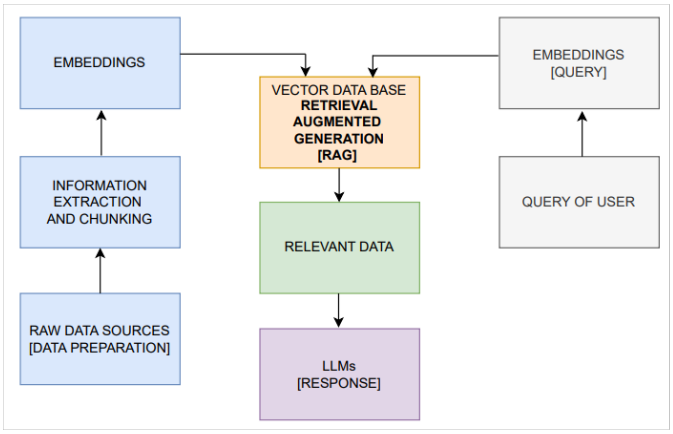
EMBEDDINGS
INFORMATION EXTRACTION AND CHUNKING
RAW DATA SOURCES [DATA PREPARATION]
VECTOR DATA BASE RETRIEVAL AUGMENTED GENERATION [RAG]
RELEVANT DATA
LLMs [RESPONSE]
EMBEDDINGS [QUERY]
QUERY OF USER

Fig. 1: Overview of the Retrieval-Augmented Generation (RAG) pipeline.

A key performance bottleneck in RAG pipelines arises from dense vector retrieval, the core operation responsible for computing similarity between high-dimensional embeddings of incoming user queries and stored database vectors. This similarity search typically involves matrix–vector or matrix–matrix operations that dominate the end-to-end latency of the system, especially as embedding dimensions scale to ranges such as 768–4096 values and database sizes grow to millions or billions of records. On conventional CPU- and GPU-based platforms, dense retrieval is frequently limited by memory bandwidth rather than computational capability, since the entire vector database must be continuously streamed from external memory. This results in extremely low operational intensity and high energy consumption, making predictable throughput difficult under strict latency, power, and memory constraints.

To address these limitations, this work presents a computationally optimized dense-retrieval architecture for Retrieval-Augmented Generation on the Xilinx Versal VCK5000 AI Engine platform. The architecture employs a batched retrieval pipeline in which the embedding database is partitioned across multiple AI Engine tiles. Each tile locally stores and processes a subset of vector entries, while incoming query vectors are broadcast to all tiles. The database is delivered through a packet-streamed dataflow mechanism that minimizes buffering overhead and maintains continuous high-throughput processing, reducing reliance on off-chip memory access.

Two dataflow strategies are implemented and evaluated in this work: an inner-product formulation based on packet-streamed database tiles and a compute-optimized outer-product formulation using k-block streaming with tile-local accumulation. Experimental evaluations show that although the inner-product approach is conceptually simple and straightforward to implement, it remains fundamentally memory-bound because large blocks of the database must be repeatedly streamed. In contrast, the outer-product design streams each k-block only once and leverages tile-local reuse to achieve significantly higher sustained computational throughput measured in GFLOP/s. This demonstrates that careful architectural co-design and dataflow restructuring can substantially improve dense retrieval efficiency.

Overall, this work highlights the importance of accelerator-optimized system design for scalable and responsive RAG deployments. The proposed architecture demonstrates how tightly coupled AI Engine compute and dataflow streaming can overcome memory bottlenecks, enabling dense retrieval engines that meet the latency, power, and predictability requirements of real-time LLM-based decision systems.

II. Related Work

Approximate nearest-neighbor (ANN) search has been widely studied as a means of scaling similarity search over large vector databases. Libraries such as FAISS (Facebook AI Similarity Search), HNSW (Hierarchical Navigable Small World graphs), and ScaNN (Scalable Nearest Neighbor) implement ANN search using a combination of clustering, graph traversal, and quantization techniques. FAISS relies on inverted file indices and product quantization to reduce the search space while HNSW builds a multi-layer small world graph enabling logarithmic time approximate search and ScaNN employs lightweight re-ranking and partitioning strategies to balance speed and recall. Although these systems achieve high throughput on CPU and GPU platforms, they often trade off exactness for speed and remain constrained by memory bandwidth when dealing with large embedding tables.

GPU-based retrieval engines exploit massive parallelism to accelerate dense matrix-vector and matrix-matrix operations through optimized GEMM kernels. While GPUs achieve excellent performance for large batched workloads, their performance for single query or small batch retrieval is commonly limited by the cost of repeatedly streaming database embeddings from global memory. This mismatch between compute capability and memory bandwidth often prevents GPUs from reaching their theoretical peak throughput for retrieval workloads.

FPGA and ASIC accelerators have explored custom compute fabrics such as systolic arrays, pipelined MAC units, and near memory compute architectures to reduce data movement overheads. These designs offer opportunities for low-latency and energy-efficient retrieval, but tend to be rigid in tiling strategies, precision modes, or dataflow configurations, limiting their generality.

AI Engine-based acceleration on platforms such as the AMD VCK5000 enables fine-grained tiling, low latency streaming between tiles, and high-throughput fused multiply and accumulate kernels. Unlike traditional ANN accelerators, the architecture supports flexible packet streams, broadcast networks, and tile-local memory, allowing designers to restructure kernels for improved data reuse. Our work differs from prior literature by demonstrating how a shift from an inner-product formulation to an outer-product formulation substantially increases operational intensity, reduces the need for repeated streaming of the vector database and ultimately transitions execution from memory-bound to compute-bound. This aligns the computation more effectively with the architectural strengths of the AIE array and leads to higher sustained throughput for dense retrieval.

III. Background

A. RAG and Dense Semantic Retrieval

Retrieval-Augmented Generation (RAG) systems enhance large language models by retrieving semantically relevant documents from an external vector database. Unlike purely parametric models, RAG decouples knowledge storage from model weights, improving factuality and reducing hallucinations. At the core of RAG retrieval lies dense semantic search, where both queries and documents are embedded into a shared vector space.

Given a normalized query embedding $q \in \mathbb{R}^d$ and a database of document embeddings $\{v_i\}_{i=1}^n$, retrieval is performed via inner-product similarity:

$$\mathrm{sim}(v_i, q) = v_i^T q,$$

which is equivalent to cosine similarity for unit-normalized vectors.

Dense retrieval therefore requires executing millions to billions of dot products per query. This computation can be expressed as a large-scale matrix–vector product, and for batched queries, as a matrix–matrix product, making the problem well suited to specialized hardware accelerators.

B. Exact vs. Approximate Search

Exact dense retrieval computes achieving the highest recall but at significant computational cost.

$$r = Vq, \quad V \in \mathbb{R}^{n \times d}$$

Approximate nearest-neighbor (ANN) methods such as Product Quantization (PQ) reduce computation and memory bandwidth, but introduce recall degradation that is undesirable in high-stakes RAG pipelines, where retrieval quality directly impacts generation accuracy.

Recent system-level studies show that on modern server-class CPUs, exact search can dominate end-to-end RAG latency, accounting for up to 97% of inference time for large databases. Hardware acceleration of exact retrieval is therefore crucial for scalable, high-quality RAG.

IV. Method and Approach

Xilinx Versal VCK5000 Architecture Overview

The AMD Versal VCK5000 is a heterogeneous compute platform consisting of three main domains: the **Processing System (PS)**, **Programmable Logic (PL)**, and **AI Engines (AIEs)**. The platform is designed to accelerate dense computation workloads such as Retrieval-Augmented Generation (RAG) with high throughput and low latency.

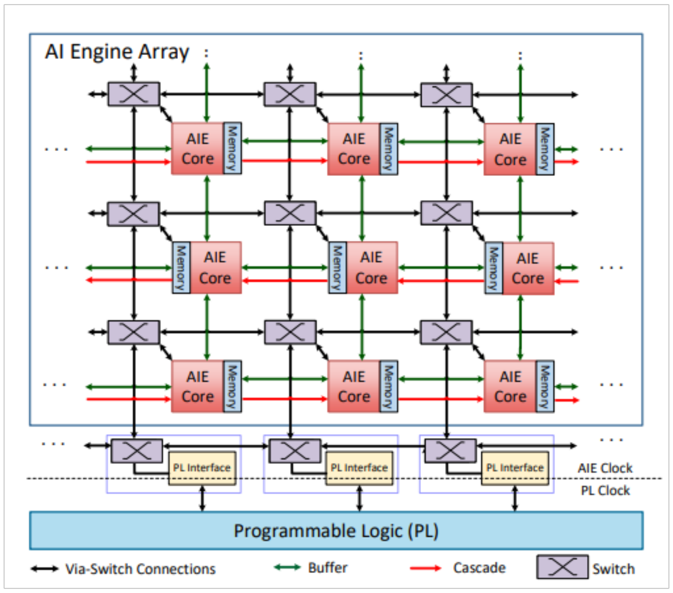

Fig. 2: Versal AIE architecture

A. Processing System (PS)

The PS includes a **dual-core ARM Cortex-A72 processor** which serves as the host for configuring and controlling the AIEs and PL. Key features include:

- Acts as a programming interface for both **AI Engines** and **Programmable Logic**.
- Capable of running a **Linux operating system**, enabling:
  - File I/O and application-level control.
  - High-level orchestration of PL and AIE workloads.
- Central command unit for the heterogeneous compute domains.

B. Programmable Logic (PL)

The PL is the FPGA-based reconfigurable fabric of the VCK5000 and includes:

- **Lookup Tables (LUTs)**, **Flip-Flops (FFs)**, **Digital Signal Processing slices (DSPs)**, **Block RAM (BRAM)**, and **UltraRAM (URAM)**.
- Supports the implementation of:
  - Custom accelerators.
  - Data movement hardware such as buffers, crossbars, and packet routing.
  - Any application-specific datapath required by workloads.

C. AI Engines (AIEs)

The AIE array consists of **vector processors operating at 1.25 GHz**, optimized for dense linear algebra operations:

- Each AIE contains:
  - 32 KB of tightly coupled local memory for instructions and data.
  - Vector processor capable of:
    - 8 FP32 MACs per cycle.
    - 128 INT8 MACs per cycle.
- Inter-AIE connectivity through switch connections, buffer paths, and cascade links.
- Well-suited for matrix multiplication and other dense compute workloads.

D. AIE Interfaces

AIEs have two main interface types:

- **NoC Interface Blocks**:
  - Connect to DRAM via the Network-on-Chip (NoC).
- **PL Interface Blocks**:
  - Connect to the Programmable Logic through the **PLIO** interface.
  - PLIO default width: 64 bits; optional: 128 bits at half the frequency.

E. Memory Hierarchy

The VCK5000 has a three-level memory hierarchy:

1) **Level 1: AIE Local Memory**:
   - Each AIE: 32 KB.
   - 400 AIEs total: 12.8 MB.
2) **Level 2: PL Memory**:
   - **BRAM**: 967 units × 36 Kbit = 4.6 MB.
   - **URAM**: 463 units × 288 Kbit = 17.1 MB.
3) **Level 3: External DRAM**:
   - 16 GB DDR4 memory on-board.
   - Accessible via NoC from both PL and AIE.

F. Programming Model

- AIE kernels are written in **C/C++**.
- Two approaches:
  - **High-Level APIs**: Predefined operations such as GEMM and FIR filters.
  - **Intrinsics-Based Programming**: Low-level architecture-specific instructions for maximum performance and parallelism control.

G. System Throughput and Bandwidth

- **AI Engine Throughput** (400 AIEs, 1.25 GHz):
  - FP32 peak: $1.25\,\text{GHz} \times 8 \times 400 \times 2 = 8$ TFLOPs.
  - INT8 peak: $1.25\,\text{GHz} \times 128 \times 400 \times 2 = 128$ TOPs.
- **PL–AIE Interface Bandwidth**:
  - Single PLIO (500 MHz × 64-bit): 4 GB/s.
  - Connections per AIE row:
    - PL$\rightarrow$AIE: 8 links $\rightarrow$ total 1.2 TB/s.

    * AIE→PL: 6 links → total 0.9 TB/s.

- **DRAM Bandwidth**:
  - Four DDR4 channels (4 GB each, 72-bit, 3200 Mb/s) → 102 GB/s total.
  - Accessible from both PL and AIE through the NoC.

V. Dense Retrieval

Dense retrieval computes the similarity between an incoming query embedding and a large vector database. A common method to obtain cosine similarity or dot-product similarity is to perform matrix–vector multiplication, where the database vectors are arranged as a matrix and multiplied by a query vector to determine the highest similarity score.

A. Memory-Bound Nature of Matrix–Vector Multiplication

If a query has to be answered based on a given vector database, simplest way would be to take the vector database as a matrix and obtain dot product between the Vector database matrix defined by V and Query vector defined by Q .
The rows of matrix V define the number of vectors N present in a vector database while the columns state the dimension of each vector N_d. The rows of Q determine the number of queries N_q. The obtained resultant dotproduct matrix R therefore has dimensions of N*1.

$$\underset{V}{\begin{bmatrix} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ v_{31} & \cdots & v_{3d} \\ \vdots & \ddots & \vdots \\ v_{n-1,1} & \cdots & v_{n-1,d} \\ v_{n1} & \cdots & v_{nd} \end{bmatrix}} \times \underset{Q}{\begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_d \end{bmatrix}} = \underset{R}{\begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_{n-1} \\ r_n \end{bmatrix}}$$

$$V \in \mathbb{R}^{n\times d}, \quad Q \in \mathbb{R}^{d\times 1}, \quad R = VQ \in \mathbb{R}^{n\times 1},$$

Matrix–vector (MV) multiplication for a single query ($B = 1$) is heavily memory-bound because the entire vector database must be streamed for each query. Each vector of dimension D is loaded from memory to compute a single dot product with the query, resulting in very few computations per byte accessed.

The operational intensity (OI) for this process is therefore very low:

$$\text{OI}_{MV} = \frac{\text{FLOPs}}{\text{Bytes transferred}} \approx \frac{2ND}{4ND} = 0.5 \text{ FLOP/Byte},$$

where N is the number of vectors, and D is the dimension of each vector. This indicating that memory transfers dominate the execution time.

As the vector database gets longer by dimension than what can be fit on a single AI engine tile , then the vector database needs to be divided into several blocks which can be sent to different AI engine tiles for computation of dot product.

$$\underset{V}{\left[\begin{array}{ccc} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ \hline v_{31} & \cdots & v_{3d} \\ \vdots & \ddots & \vdots \\ \hline v_{n-1,1} & \cdots & v_{n-1,d} \\ v_{n1} & \cdots & v_{nd} \end{array}\right]} \times \underset{Q}{\begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_d \end{bmatrix}} = \underset{R}{\left[\begin{array}{c} r_1 \\ r_2 \\ \hline r_3 \\ \vdots \\ \hline r_{n-1} \\ r_n \end{array}\right]}$$

After the computation of each block , all blocks need to be compared for the largest cosine similarity and its respective index.

Consequently, the sustained performance of MV-based single-query retrieval collapses to approximately 0.02 GFLOPs/sec on high-frequency processors, as computation is entirely gated by memory bandwidth. This limitation is alleviated by batching queries i.e. through Matrix-Matrix Multiplication, which increases operational intensity through improved reuse of the vector database.

B. Matrix-Matrix Multiplication

If there are number of queries , they can be batched to form a matrix Q whose rows represent the number of queries N_q while the columns represent the dimensions of queries D_q.Now the obtained resultant dot product matrix R obtained between the vector database matrix V and Q have the dimensions of N*D_q.

$$V = \begin{bmatrix} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ v_{31} & \cdots & v_{3d} \\ \vdots & \ddots & \vdots \\ v_{n-1,1} & \cdots & v_{n-1,d} \\ v_{n1} & \cdots & v_{nd} \end{bmatrix} \qquad Q = \begin{bmatrix} q_{11} & \cdots & q_{1q} \\ q_{21} & \cdots & q_{2q} \\ \vdots & \ddots & \vdots \\ q_{d1} & \cdots & q_{nq} \end{bmatrix}$$

$$R = \begin{bmatrix} r_{11} & \cdots & r_{1q} \\ r_{21} & \cdots & r_{2q} \\ r_{31} & \cdots & r_{3q} \\ \vdots & \ddots & \vdots \\ r_{n-1,1} & \cdots & r_{n-1,q} \\ r_{n1} & \cdots & r_{nq} \end{bmatrix}$$

$$V \in \mathbb{R}^{n\times d}, \quad Q \in \mathbb{R}^{d\times q}, \quad R = VQ \in \mathbb{R}^{n\times q},$$

In case of Longer Vector Data base, it can be divided into several blocks which can be sent to different AI engine tiles for computation of dot product.

$$V = \left[\begin{array}{ccc} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ \hline v_{31} & \cdots & v_{3d} \\ \vdots & \ddots & \vdots \\ \hline v_{n-1,1} & \cdots & v_{n-1,d} \\ v_{n1} & \cdots & v_{nd} \end{array}\right] \qquad Q = \begin{bmatrix} q_{11} & \cdots & q_{1q} \\ q_{21} & \cdots & q_{2q} \\ \vdots & \ddots & \vdots \\ q_{d1} & \cdots & q_{dq} \end{bmatrix}$$

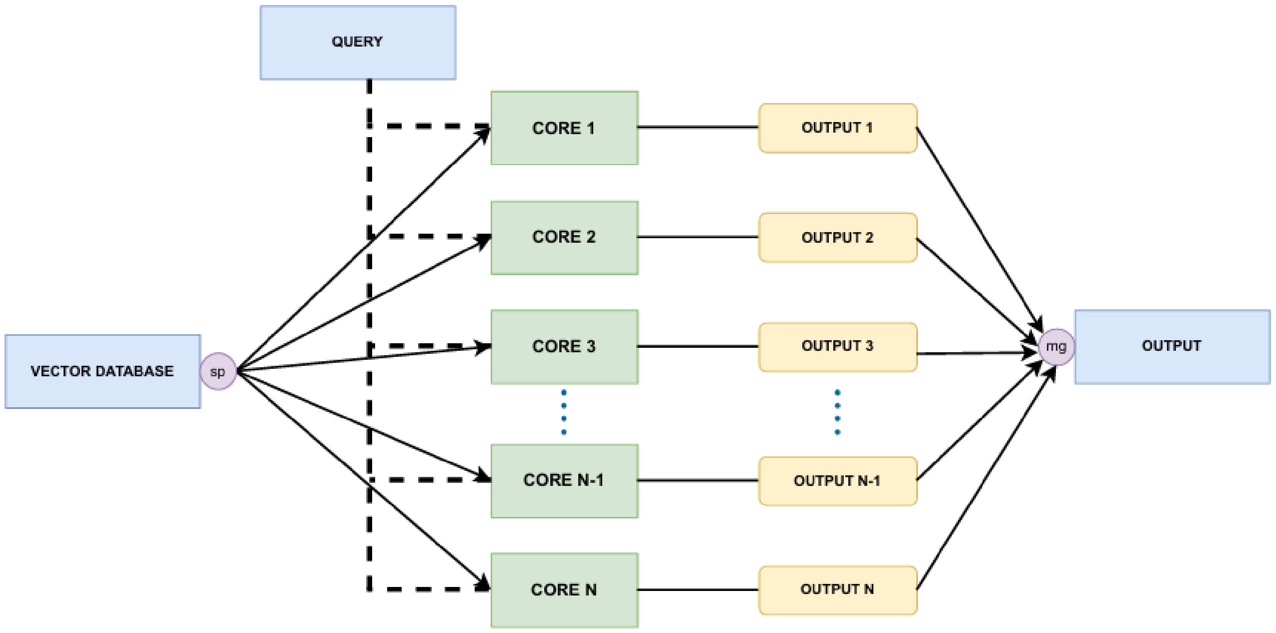

Fig. 3: Packetstreaming Vector DataBase and Broadcasting Query

$$R = \left[\begin{array}{ccc} r_{11} & \cdots & r_{1q} \\ r_{21} & \cdots & r_{2q} \\ \hline r_{31} & \cdots & r_{3q} \\ \vdots & \ddots & \vdots \\ \hline r_{n-1,1} & \cdots & r_{n-1,q} \\ r_{n1} & \cdots & r_{nq} \end{array}\right]$$

After the computation of each block , all blocks need to be compared for the largest cosine similarity and its respective index just like Matrix-Vector computation.

In case of Matrix-Matrix Multiplication, the matrices are not consumed as full two-dimensional arrays. Instead, the inputs are partitioned into fixed-size tiles that match the compute unit architecture. Although the matrices appear to be two-dimensional, they are stored in memory as a flattened one-dimensional sequence. Kernel loads and reuse these blocks efficiently. Tiling and tile-wise memory layout greatly improve cache locality and enable high-throughput matrix-matrix multiplication.

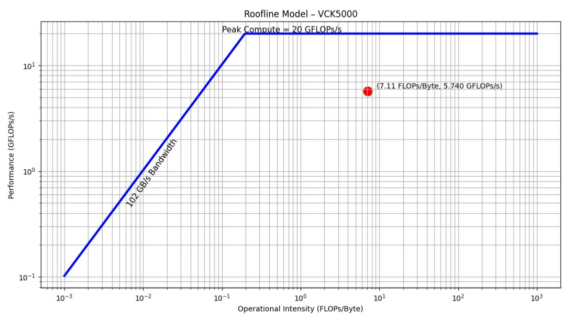

Fig. 4: Roofline Analysis for single Kernel

For a Single Tile:
Peak Compute possible:

$$FP32_{thrpt} = AIE_{freq} \times PeakThrpt_{FP32} \times 2(*,+) = 1.25\,GHz \times 8 \times 2$$

$$= \mathbf{20\ GFLOPS/s}$$

Compute for a Single Tile:
For a Matrix Matrix Multiplication with Vector database as 128x32 matrix and Query matrix as 32x32 :
Flops = 262144
Cycles = 56993 (with tiling as 4x2x4)
Compute at 1.25 GHz = 5.74 GFlops/s
Operational Intensity for a Single Tile: 7.11 Flops/Byte
Dataset used is available at Google Natural Questions
The roofline plot in Fig. 4 places the measured performance of our kernels in the context of the hardware's theoretical compute and memory bandwidth limits. Points lying on the left of the ridge indicate memory-bound execution, whereas points near the flat roof indicate compute-bound behavior. Our single-tile inner-product kernel falls on the right sde with throughput of 20GFlops/s and OI of 7.11Flops/Byte. In order to increase the compute we need to increase the Data Reuse happening.

C. Packet Stream of Vector Database and Broadcasting the Query

In order to perform Co-sine similarity between Query and Huge Vector Database, then the Vector DataBase is split into multiple tiles, using packet stream function while the Query is broadcasted across all tiles. The performance is obtained as shown in 3.

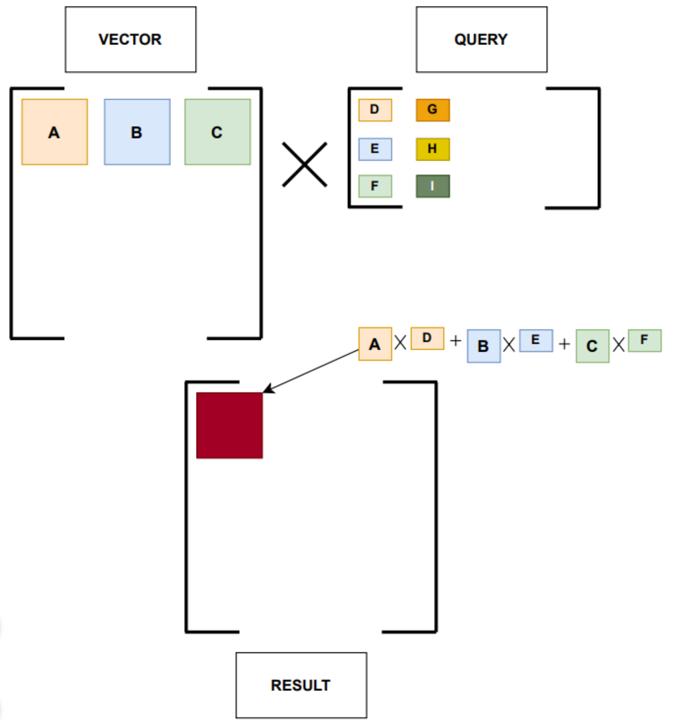

Fig. 5: Inner Product Flow

D. Inner-Product vs. Outer-Product Methods for Matrix–Matrix Multiplication

1) Inner-Product Method (Packet-Stream DB, Broadcast Query): In the inner-product formulation, the vector database V is supplied as a packet stream to each tile, while the query matrix Q is broadcast to all tiles. This enables efficient reuse of Q, since the same query data reaches every tile without additional memory traffic. Each tile performs partial dot-product computations for its assigned region of the output. However, the database V must be re-streamed whenever a tile needs the same K-dimension data again. Because tiles cannot retain all reused portions of V locally across compute phases, database elements are fetched repeatedly. This repeated streaming limits data reuse, reduces operational intensity, and results in lower sustained throughput. The method naturally produces column-oriented partial results, which must be combined across tiles to obtain global column maxima. **Pseudocode**

Algorithm 1 Inner Product

```
0: broadcast $Q \rightarrow$ all tiles
0: for each tile do
0:     load local $V_{\text{tile}}$ from pktstream
0:     $C_{\text{tile}} \leftarrow 0$
0:     for each k-block do
0:         $C_{\text{tile}} \leftarrow C_{\text{tile}} + \text{dot\_product}(Q[k], V_{\text{tile}}[k])$ {MMUL mul+mac}
0:     end for
0:     compute tile_local_column_max
0: end for
0: reduce tile_local_column_max $\rightarrow$ global_column_max =0
```

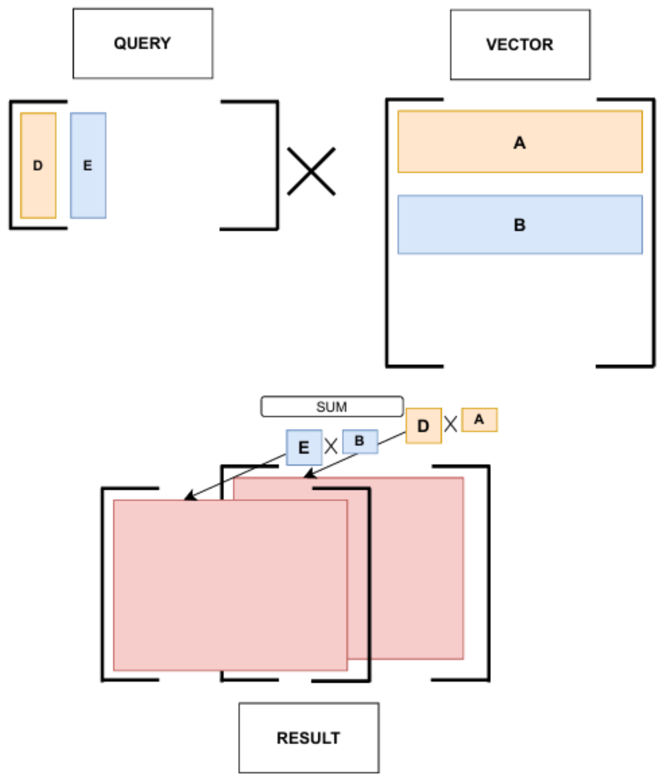

Fig. 6: Outer Product Flow

2) Outer-Product Method (k-Block Streaming and Accumulation): In the outer-product formulation, both V and Q are streamed in aligned k-blocks. For each k-block, tiles simultaneously receive the broadcast portion of Q and the corresponding packet-stream portion of V, allowing each tile to compute a partial outer-product update. These partial results are accumulated into a tile-local buffer C_{accum}.

Because every k-block of V is consumed exactly once and reused across all rows and columns of the tile during accumulation, this strategy achieves much stronger data reuse and significantly higher operational intensity compared to the inner-product method. The accumulated structure also aligns naturally with row-oriented reductions, enabling each tile to produce row-wise maxima before final global reduction.
Pseudocode

Algorithm 2 Row-Max Reduction Across Tiles

```
0: $C_{\text{accum}} \leftarrow 0$
0: for each k-block do
0:     read $Q_k$ from broadcast
0:     read $V_k$ from pktstream
0:     for each tile do
0:         $C_{\text{accum}} \leftarrow C_{\text{accum}} + \text{outer\_product}(Q_k, V_k)$ {MMUL mul}
0:     end for
0: end for
0: compute tile_local_row_max
0: reduce tile_local_row_max $\rightarrow$ global_row_max =0
```

While this approach removes the repeated streaming overhead inherent to the inner-product method, it requires tile-local storage for C_{accum} and introduces a more structured scheduling pattern due to k-block sequencing. These trade-offs are offset by the improved throughput and scalability offered by the outer-product formulation.

Motivation for the Transition: Although straightforward, the inner-product approach suffers from repeated streaming of large database tiles, which restricts performance on memory-bound workloads. The outer-product method avoids this limitation by streaming each k-block of V only once and reusing it fully within each tile through accumulation into C_{accum}. This substantially increases operational intensity and sustained FLOP/s, making the outer-product formulation more efficient and scalable for large matrix–matrix multiplication workloads.

Solution Pipeline Overview: The end-to-end retrieval pipeline deployed on the VCK5000 follows a structured dataflow across the Programmable Logic (PL), the AI Engine (AIE) array, and the tile-level compute kernels. The vector database is first packetized in the PL and streamed into the AIE array as a packet stream, while the query matrix is broadcast so that identical query blocks reach all tiles concurrently. This arrangement ensures minimal latency for query distribution and enables each tile to operate independently on its assigned partition of the database.

The inner-product formulation was initially chosen as the baseline because it maps directly to standard GEMM execution and allows reuse of the already optimized single-tile MMUL kernel. It provides a simple and correct baseline implementation while exposing the memory-related limitations of streaming the large vector database. Transitioning to the outer-product formulation addresses exactly these limitations. By streaming the database in k-blocks and accumulating partial outer products inside each tile, the method ensures that every database block is consumed once and reused fully. This increases operational intensity, reduces memory bandwidth pressure, and improves sustained throughput.

Across both versions, optimizations focus on: (i) tiling matrices to match the $4 \times 2 \times 4$ MMUL configuration, (ii) storing intermediate buffers in static tile-local memory to avoid stack overflows, (iii) using packet streams and broadcasts efficiently, and (iv) reducing intermediate results locally before global aggregation. The primary performance metrics targeted are sustained GFLOP/s, operational intensity, memory traffic, and scalability across tiles.

VI. Results and Discussion

The GFLOP/s comparison highlights the performance differences among the three platforms—CPU, GPU, and VCK5000. The CPU baseline achieves only 0.0325 GFLOP/s, mainly due to sequential execution and Python-loop overhead, which severely limits arithmetic throughput.

The GPU (NVIDIA T4) reaches 12.42 GFLOP/s, benefiting from highly optimized parallel compute units and cuBLAS GEMM operations. However, the relatively small problem size (32×128×32) prevents the GPU from achieving higher throughput, as kernel launch and memory overhead dominate compared to useful computation.

The VCK5000 implementation achieves 8.06 GFLOP/s, which is significantly faster than the CPU and approaches the GPU's throughput. This performance is enabled by the Versal ACAP architecture, where deeply pipelined dataflow, parallel execution across AIE tiles, and deterministic scheduling allow the workload to sustain high utilization. Unlike the GPU, the VCK5000 does not suffer large kernel-launch overheads, making it more efficient for small and medium-size compute tasks.

Overall, the results show that while the GPU achieves the highest observed throughput, the VCK5000 offers a compelling balance of performance, determinism, and architectural efficiency. Its ability to maintain high utilization on smaller workloads makes it well-suited for applications requiring predictable latency, dataflow-style computation, or tightly coupled compute and memory pipelines, as found in many AI-engine and signal-processing workloads.

Performance(GFLOP/s) for inner product is calculated as

$$\left(\frac{3 \times 128 \times 32 \times 32}{121870}\right) \times 1.25\,\text{GHz} = 8.06 GFLOPS/s \quad (1)$$

TABLE I: Performance Comparison Across CPU, GPU, and VCK5000[Our Model]

Metric	CPU	GPU (T4)	Our model
Performance (GFLOP/s)	0.0325	12.42	8.06

VII. Future Works

- **Scaling Packet-Stream and Broadcast-Based Inner-Product to All 400 Tiles:** Extending the current packet-stream and query-broadcast mechanism to the full 400-tile array is expected to substantially improve data reuse and increase sustained throughput.
- **Evaluation with a Larger Vector Database:** Testing the system with significantly larger vector databases will enable a deeper analysis of data-reuse behavior and its impact on performance, particularly on overall throughput.
- **Multi-Tile Outer-Product Execution:** Generalizing the outer-product formulation across multiple tiles can reduce the total number of operations and exploit tile-local accumulation more effectively, yielding substantial improvements in performance.
- **INT8 vs FP32 Computation Throughput:** Investigating low-precision execution (INT8) compared to FP32 can reveal significant speedups. Since INT8 operations allow the AI Engine to pack and execute many more multiply–accumulate (MAC) operations per cycle, the system can achieve substantially higher arithmetic throughput and improved energy efficiency. Future work includes implementing INT8 quantization, evaluating accuracy–performance trade-offs, and measuring end-to-end throughput gains.

- **Extended Retrieval Features:** Research on hybrid retrieval approaches, combining approximate nearest-neighbor filtering with exact re-ranking, and support compressed embeddings (PQ, HNSW-based acceleration)
- **Testing on Hardware**: Evaluate the performance of the approaches on Hardware instead of limiting it to Hardware Emulation
- **End-to-End RAG Integration:** Integrate the accelerated retrieval engine with complete RAG inference stacks to measure real response-time impact and improve system-level optimization.

We believe that these future works will make a huge difference in terms of Performance (i.e. Throughput) and utilize the Tile to the maximum whichever was limited in before sections.

VIII. Acknowledgment

The authors would like to sincerely thank Dr. Suresh Purini and Soumitra Ghosh for their valuable guidance, support, and insightful technical discussions throughout the course of this work.

IX. Conclusion

In this work, we addressed the performance bottleneck in dense retrieval within Retrieval-Augmented Generation (RAG) systems, where the cost of repeatedly streaming large vector databases leads to memory-bound execution and low operational intensity on traditional CPU and GPU platforms. To overcome these limitations, we developed an optimized dense-retrieval architecture on the Xilinx Versal VCK5000 AI Engine array, reformulating similarity search using matrix–vector and matrix–matrix multiplication with tiled compute kernels. Our approach partitions the embedding database across multiple AI Engine tiles and evaluates two key dataflow strategies: an inner-product formulation and a more compute-efficient outer-product formulation.

Experimental evaluation demonstrates that the inner-product baseline remains limited by repeated database streaming, restricting throughput, whereas the outer-product approach streams each k-block once, performs tile local accumulation, and significantly enhances data reuse. As a result, the design achieves 8.06 GFLOP/s on the VCK5000 far outperforming the CPU baseline of 0.0325 GFLOP/s and approaching GPU throughput of 12.42 GFLOP/s showing a substantial improvement in sustained performance and efficiency. These results highlight that algorithm architecture co-design, rather than incremental micro-optimizations, is essential to unlocking high utilization of the Versal AI Engine array.

In general, this study demonstrates that reorganizing dense retrieval around optimized dataflow and tiling strategies is crucial for scalable and low-latency RAG execution. The proposed architecture delivers deterministic performance and improved operational intensity, making it a promising direction for real-time enterprise-scale RAG systems. Future extensions across multi-tile scaling and larger datasets offer potential for further acceleration and system-level impact.

Our analytical model can be used by researchers and developers. This model, along with our code and experimental setup, is available for use at: **Link**

References

[1] Product Quantization for Nearest Neighbor Search, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 33, no. 1, pp. 117-128, Jan. 2011

[2] Dense Passage Retrieval for Open-Domain Question Answering, Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), November 2020

[3] Accelerating Retrieval-Augmented Generation, publication at ASPLOS 2025

[4] AMD Vitis Tutorial: AI Engine Development – Packet Switching, AMD Documentation, Release 2022.2