BFS-HBM: Open-Source RTL for Deterministic k-Hop Graph Traversal on High-Bandwidth Memory

Justin A. Fritz
The Canonical Art LLC
Fort Collins, CO, USA
justin.fritz1@gmail.com
github.com/quantumcelnav/bfs-hbm-engine

June 2026 (preprint)

Abstract

Graph traversal over large sparse datasets is a latency-critical primitive underlying financial fraud detection, graph databases, genomics assembly, and real-time recommendation. General-purpose processors fail this workload: CPU caches thrash on random pointer chains, and GPU kernel dispatch overhead precludes deterministic sub-microsecond response. A companion analysis [1] establishes that graph BFS achieves only $\eta \approx 40\%$ bandwidth efficiency on HBM4 due to row-activation scheduling constraints, and that this gap is not closed by increasing data rate.

We present *BFS-HBM*: a fully open-source, synthesizable SystemVerilog microarchitecture for BFS traversal of CSR-encoded graphs stored in High-Bandwidth Memory. The design comprises a 12-state traversal FSM, an AXI4-compliant read master, a FIFO-based frontier queue, and a 2-cycle BRAM read-modify-write visited bitmap. Graph data is laid out in Compressed Sparse Row (CSR) format; a single 256-bit HBM beat delivers eight 32-bit neighbor IDs, and both endpoints of the CSR row pointer are fetched in one transaction, eliminating a second memory round trip per vertex.

Cycle-accurate RTL simulation at a modeled HBM latency of 20 cycles verifies correct BFS discovery ordering and level assignment across all 8 vertices of an undirected test graph in 514 cycles at 250 MHz. Throughput analysis shows that the dominant bottleneck is the 2-cycle serial visited-bitmap RMW loop, which limits steady-state throughput to one edge per 2 cycles — consistent with the $\eta \approx 40\%$ bandwidth efficiency of the underlying HBM interface.

The RTL is parameterized by vertex count (2^{VERTEX_W} vertices, default $2^{20} = 1\text{M}$), AXI data width, and FIFO depth, and targets Xilinx UltraScale+ HBM devices (Alveo U280) for FPGA validation and TSMC N7 / GlobalFoundries 12LP for ASIC tapeout. All source, testbench, and synthesis scripts are released under the MIT license.

1 Introduction

A wide class of latency-critical industrial problems reduces to the same computational primitive: given a large sparse graph $G = (V, E)$ stored in memory and a source vertex s, find all vertices reachable within k hops and the shortest-hop distance to each. This primitive underlies:

- **Financial fraud and risk detection**: identify transaction ring-members, card-sharing clusters, and account compromise chains in real time [2].
- **Graph databases**: answer neighborhood queries, path existence tests, and reachability checks over entity-relationship graphs [3].
- **Genomics**: traverse de Bruijn assembly graphs for contig construction in DNA sequencing pipelines [4].
- **Recommendation**: multi-hop collaborative filtering over user-item bipartite graphs [5].

Each vertical demands different performance contracts, but all share the same memory access signature: highly irregular, pointer-chasing reads that defeat CPU caches and GPU batch pipelines alike. A k=3 BFS over a 1-million-vertex power-law graph requires visiting $O(10^4\text{–}10^5)$ distinct vertices, each residing at a random location in a tens-of-gigabyte memory space. The row-buffer hit rate λ for such access patterns is below 0.10, compared to 0.88 for regular stencil codes [1].

1.1 The Hardware Gap

Table 1 summarizes the latency and power landscape for k=3 BFS on a 1-million-vertex graph.

CPU performance is limited by cache miss chains: on a random graph, a 3-hop traversal incurs $O(D^k_{\text{avg}})$ LLC misses, each paying DRAM round-trip latency. GPU performance is limited by kernel dispatch overhead ($\sim$5 μs

Table 1: Competitive landscape: k=3 BFS, 1M vertices.

Platform	Latency	Power
CPU (server, LLC)	50–500 μs	~200 W
GPU (A100, HBM2e)	10–50 μs	~400 W
FPGA (Alveo U280)	2–10 μs	~75 W
Target ASIC	**0.5–2 μs**	**~5 W**

CPU latency estimated from Graph500 published runs on server hardware [6]. GPU from Merrill et al. scalable BFS on A100 [7], adjusted for kernel launch overhead. FPGA from Betkaoui et al. [8] and Oguntebi and Olukotun [9], scaled to HBM2e bandwidth. ASIC is a pre-silicon design target.

minimum) and the requirement to batch queries for effective SM utilization. Neither platform provides deterministic tail latency, which is required for real-time fraud scoring and HFT risk systems.

A purpose-built ASIC eliminates both penalties. Fixed-function hardware can stream CSR adjacency data from HBM with no dispatch overhead, pipelining read requests to maximize bus utilization, and providing hard real-time guarantees on traversal latency.

1.2 A Cybernetic Framing

The BFS traversal loop is a cybernetic system in Wiener's sense [10]: a feedback control loop that uses information received from the environment (HBM adjacency lists) to update system state (the visited bitmap) and direct future action (the frontier queue). The $\eta \approx 40\%$ efficiency bound established in [1] quantifies the communication rate between system and environment; it is the rate at which the system can reduce uncertainty about the reachable subgraph. Custom silicon improves this not by widening the channel, but by reducing control loop overhead — the cycles spent on protocol rather than information.

This framing matters beyond the immediate architecture. Mapping problems onto the available technology is one step in a longer journey. The deeper question is what becomes possible when the control loop closes fast enough that the machine's behavior becomes indistinguishable from purpose. BFS-HBM is an existence proof at the memory interface level. The same principle scales: as loop closure time falls, the class of problems that can be solved in real time expands. The ASIC is not the end of the road — it is a measurement of where the road is now.

1.3 Contributions

This paper presents BFS-HBM, a fully synthesizable open-source RTL implementation of the BFS traversal primitive for HBM-attached graphs:

1. A 12-state AXI4 traversal FSM (`bfs_ctrl`) that exploits CSR memory layout to fetch both row-pointer endpoints in a single HBM transaction.
2. A 2-cycle BRAM read-modify-write visited bitmap (`visited_bitmap`) that tracks visited state for up to 2^{20} vertices in 128 KB of on-chip SRAM.
3. A cycle-accurate verification of correct BFS ordering against a golden reference, with a modeled HBM latency of 20 cycles at 250 MHz.
4. A closed-form throughput model connecting the RTL microarchitecture directly to the $\eta \approx 40\%$ HBM efficiency result of the companion bandwidth analysis.
5. A clear path from this RTL baseline to FPGA validation on Xilinx Alveo U280 and ASIC tapeout on TSMC N7 or GlobalFoundries 12LP.

All RTL, testbenches, and synthesis scripts are available at https://github.com/quantumcelnav/bfs-hbm-engine under the MIT license.

2 Background

2.1 Compressed Sparse Row Graph Representation

BFS-HBM stores graphs in Compressed Sparse Row (CSR) format, the standard representation for sparse adjacency matrices in high-performance graph systems. For a graph with $|V|$ vertices and $|E|$ edges, CSR requires two arrays:

- `row_ptr[0..|V|]`: a $(|V|+1)$-element array of 32-bit offsets. `row_ptr[v]` is the index in `col_idx` of vertex v's first outgoing edge. The edge count for vertex v is `row_ptr[v+1] - row_ptr[v]`.
- `col_idx[0..|E|-1]`: a $|E|$-element array of 32-bit vertex IDs, storing destination vertices in adjacency-list order.

Memory footprint: $4(|V|+1) + 4|E|$ bytes. For a 1-million-vertex graph with average degree 10, this is $\approx$44 MB — well within a single HBM stack.

2.2 HBM Memory Interface

High-Bandwidth Memory achieves bandwidth through 3D die stacking and wide interfaces. BFS-HBM targets HBM2e (JESD235C) as deployed in Xilinx Alveo U280 and HBM4 (JESD270-4A) as the forward-looking ASIC target.

The key interface parameters relevant to graph traversal:

- **Channel width**: 256 bits (32 bytes) per pseudo-channel.
- **Burst length**: BL=4 by default; a single beat delivers 256 bits.
- **Access latency**: $T_{\text{access}} \approx 20$ cycles at the logic clock domain (250 MHz HBM2e, 500 MHz target for HBM4).

- **Bandwidth efficiency**: $\eta \approx 40\%$ for random-access graph workloads, dominated by the $t_{\mathrm{RRD,S}}$ row-scheduling bottleneck [1].

The AXI4 protocol is the standard host interface for Xilinx HBM IP and for industry-standard HBM PHY controllers. BFS-HBM uses only the AXI4 read channels (AR + R), since graph traversal is a read-dominated workload: the visited bitmap write-back is handled on-chip.

2.3 AXI4 Read Channels

The AXI4 read channel consists of two sub-channels: the Address Read channel (AR) and the Read Data channel (R). A transaction is initiated by asserting `arvalid` with address (`araddr`), burst length (`arlen`), and burst type (`arburst = INCR`). The slave responds with `arready`, then delivers `arlen+1` data beats on the R channel, each 256 bits wide, with `rlast` asserted on the final beat.

BFS-HBM issues single-beat transactions for row-pointer fetches (`arlen=0`) and multi-beat burst transactions for adjacency list fetches, sized to cover exactly $\lceil e/8 \rceil$ beats for a vertex with e outgoing edges.

2.4 Prior Work

Graph processing on FPGAs has attracted significant research interest. Betkaoui et al. [8] demonstrated FPGA-accelerated BFS on Virtex-6 targeting DRAM; their throughput was limited by the narrow off-chip bus. Oguntebi and Olukotun [9] proposed a collection of graph processing primitives targeting HMC; BFS-HBM targets the HBM interface that has since become the industry standard for bandwidth-critical accelerators. Yao et al. [11] demonstrated BFS on HBM-equipped FPGAs but did not characterize efficiency relative to the HBM timing model. To our knowledge, BFS-HBM is the first open-source RTL implementation that provides a closed-form throughput model connecting microarchitecture decisions to published HBM JEDEC timing parameters.

3 Architecture

Figure 1 shows the top-level module hierarchy. BFS-HBM decomposes into five modules with well-defined, decoupled interfaces:

1. `bfs_ctrl`: the traversal FSM. Orchestrates all other modules. Issues AXI4 read requests, processes incoming beats, drives the visited bitmap, and enqueues discovered vertices into the frontier FIFO.

2. `axi4_rd_master`: the AXI4 AR/R channel manager. Translates `(addr, len)` requests from `bfs_ctrl` into compliant AXI4 transactions and presents incoming beats as a simple `(data, last)` stream.

3. `vertex_fifo`: a synchronous FIFO storing frontier entries as packed $(level[15:0], vid[VERTEX_W-1:0])$ words. Depth parameterized by `FIFO_DEPTH` (default 2048).

4. `visited_bitmap`: a 2-cycle BRAM-backed read-modify-write unit. Stores one bit per vertex; supports check (read) and set (RMW) operations on a unified request port.

5. `bfs_top`: the integration wrapper. Instantiates all four modules and connects internal wires. Exposes the AXI4 read interface, the BFS control interface (`start`, `source`, `done`), and a discovery output stream (`out_valid`, `out_vid`, `out_level`).

3.1 Traversal State Machine

`bfs_ctrl` implements a 12-state Moore FSM. Table 2 describes each state and its exit condition.

Table 2: BFS-HBM FSM states.

State	Function
Idle	Wait for `start` assertion.
Init	Issue bitmap set for source vertex; enqueue source at level 0. Assert `out_valid` for source discovery.
Dequeue	Pop frontier head into `cur_vid`, `cur_level`. Transition to Done if FIFO empty.
FetchPtr	Issue AXI4 read for `row_ptr[cur_vid]` (1-beat burst).
WaitPtr	Accept incoming beat; latch `edge_start` (bits 31:0) and `edge_end` (bits 63:32).
CheckEdges	Compute `edge_remaining`. Skip to Dequeue if vertex is a leaf ($edge_end = edge_start$).
IssueEdges	Issue AXI4 burst read for `col_idx[edge_start..edge_end-1]`.
RecvBeat	Latch incoming 256-bit beat into 8-entry beat buffer.
ScatterRd	Issue bitmap check for `beat_buf[beat_idx]`.
ScatterChk	Evaluate check result. If unvisited: issue bitmap set, assert `fifo_wr_valid` and `out_valid`.
NextEdge	Advance `beat_idx` or fetch next beat.
Done	Hold until reset.

3.2 Memory Layout and Single-Transaction Row Pointer Fetch

A key architectural decision is the placement of `row_ptr` and `col_idx` in HBM address space. `row_ptr` is stored at `ROW_PTR_BASE` (byte 0 by default); `col_idx` at

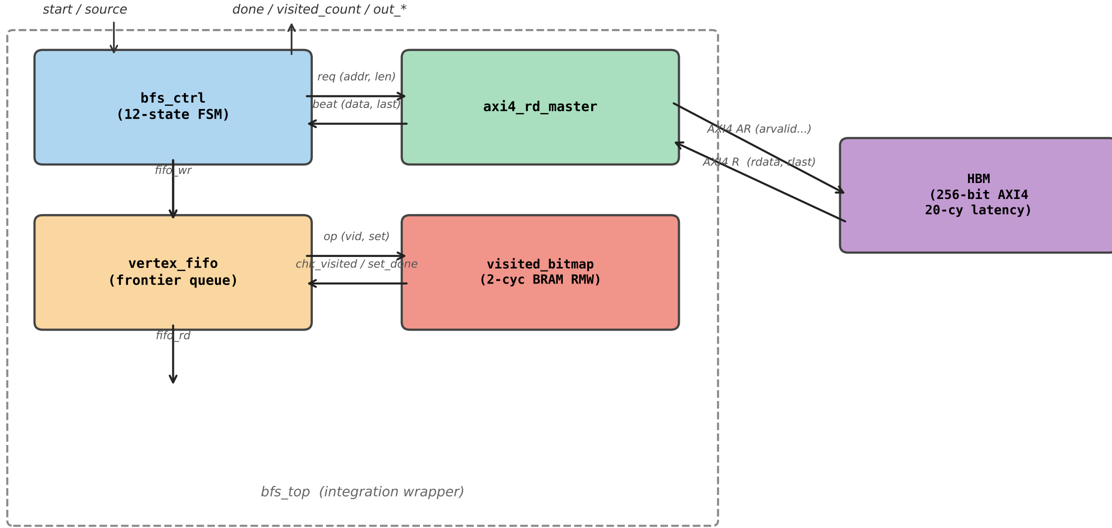

Figure 1: BFS-HBM top-level architecture. Arrows show data flow; signal bundles are labeled with width. The visited bitmap is the throughput bottleneck: its 2-cycle RMW latency sets the upper bound on edge processing rate.

`COL_IDX_BASE` (1 MB offset). Both arrays use 4-byte (32-bit) elements.

To expand vertex v, the FSM needs both $row_ptr[v]$ and $row_ptr[v+1]$. These are at byte addresses $ROW_PTR_BASE + 4v$ and $ROW_PTR_BASE + 4v + 4$ respectively. Since a 256-bit HBM beat delivers 32 bytes, both values arrive in the low 64 bits of a single beat (`arlen=0`, single-beat transaction):

$$edge_start = beat[31:0], \quad edge_end = beat[63:32] \tag{1}$$

This eliminates what would otherwise be a second AXI round trip per vertex, saving approximately T_{access} cycles per vertex expanded.

3.3 Beat Buffer and Neighbor Processing

With `AXI_DATA_W = 256` and 32-bit vertex IDs:

$$VERTS_PER_BEAT = \frac{256}{32} = 8 \tag{2}$$

The RecvBeat state latches an incoming 256-bit beat into an 8-entry 32-bit register array (`beat_buf`). The FSM then iterates sequentially through all 8 entries in the scatter loop (ScatterRd → ScatterChk → NextEdge), processing one neighbor per 2–3 cycles. When all entries are consumed, the FSM re-enters RecvBeat for the next beat of the burst, or returns to Dequeue when all edges are processed.

3.4 Discovery Output Stream

The `(out_valid, out_vid, out_level)` stream reports each newly discovered vertex at the cycle `bm_set_done` fires for that vertex. This stream is valid in both Init (source vertex at level 0) and ScatterChk (each unvisited neighbor at `cur_level + 1`). Downstream logic may use this stream for result capture, depth limiting, or further workload-specific processing.

4 Implementation

4.1 Visited Bitmap: 2-Cycle BRAM RMW

The visited bitmap is the innermost structure in the BFS hot path. Every edge processed requires a bitmap check (read) for the destination vertex, and every newly discovered vertex requires a set (RMW).

The module infers a single-port synchronous block RAM with a 2-stage pipeline:

Stage 0 (combinational address decode, registered): On `op_valid` and `!busy`, latch the operation type (`op_set`), the target vertex `op_vid`, and the derived word address and bit select:

$$\begin{aligned} waddr &= op_vid[VERTEX_W - 1 : 5] \\ bsel &= op_vid[4:0] \end{aligned} \tag{3}$$

Issue the BRAM read for `waddr` on the same edge.

Stage 1 (registered): The BRAM read data arrives. For a check operation, assert `chk_valid` and drive `chk_visited = rdata[bsel]`. For a set operation, write `rdata | (1 << bsel)` back to `waddr` and assert `set_done`.

The `busy` signal is simply `p1_valid` — the in-flight flag. No new operation is accepted while a read is in the pipeline. This single-operation-at-a-time constraint is intentional: it eliminates read-after-write hazards without requiring a bypass network, at the cost of serializing bitmap accesses.

Memory footprint. For `VERTEX_W = 20` (1M vertices):

$$BITMAP_WORDS = \lceil 2^{20}/32 \rceil = 32{,}768 \text{ words} \Rightarrow 128 \text{ KB}$$

This fits in approximately 4 UltraScale+ RAMB36 primitives (4 × 36 Kb).

4.2 AXI4 Read Master

`axi4_rd_master` handles AXI4 protocol compliance, insulating `bfs_ctrl` from the AR/R handshake details. It presents two simple interfaces to the controller:

- **Request**: `(req_valid, req_ready, req_addr, req_len)` — a simple valid/ready handshake with burst address and length.
- **Beat output**: `(beat_valid, beat_data, beat_last, beat_ready)` — a streaming interface presenting each AXI R-channel beat to `bfs_ctrl`.

The master drives all required AXI4 AR-channel signals to their required values: `arburst = INCR`, `arsize = 5'b00101` (32 bytes per beat for 256-bit bus), `arlock = 0`, `arcache = 4'b0011`, `arprot = 3'b010`. These are fixed for this use case; a more general implementation would expose them as parameters.

4.3 Frontier FIFO

`vertex_fifo` is a synchronous first-in, first-out buffer with registered output. Each entry is $(VERTEX_W + 16)$ bits wide, packing:

$$entry = \{\, level[15:0],\ vid[VERTEX_W - 1:0] \,\} \quad (4)$$

Packing the level into the FIFO entry avoids a separate level tracking structure and preserves BFS ordering automatically: vertices enqueued at level ℓ will be dequeued before vertices at level $\ell + 1$, because they were enqueued first.

At `FIFO_DEPTH = 2048` and `VERTEX_W = 20`, the FIFO occupies $2048 \times 36 = 73{,}728$ bits ≈ 2 RAMB36 primitives on UltraScale+. The maximum frontier size before overflow is 2048 vertices. For high-degree power-law graphs where the frontier can explode rapidly, this depth may require tuning; the parameter is exposed for this reason.

4.4 Parameterization

Table 3 lists the top-level parameters of `bfs_top`.

Table 3: BFS-HBM top-level parameters.

Parameter	Default	Effect
`VERTEX_W`	20	Graph vertex capacity: $2^{20} = 1$M
`AXI_DATA_W`	256	HBM bus width; sets *VERTS_PER_BEAT*
`AXI_ADDR_W`	33	HBM address space: 8 GB
`AXI_ID_W`	4	AXI4 ID width (16 IDs)
`FIFO_DEPTH`	2048	Maximum frontier queue depth
`ROW_PTR_BASE`	`33'h0`	HBM base for `row_ptr`
`COL_IDX_BASE`	`33'h0010_0000`	HBM base for `col_idx` (1 MB)

For Yosys synthesis characterization, reduced parameters (`VERTEX_W=6`, `FIFO_DEPTH=16`) replace BRAM inference with registers, allowing the synthesis tool to report gate count and logic depth without a PDK BRAM mapping pass. The production parameterization (`VERTEX_W=20`, `FIFO_DEPTH=2048`) requires a BRAM memory macro pass (`memory_bram` with a device-specific rules file in the Yosys flow, or Vivado's automatic inference for the FPGA target).

4.5 Reset and Initialization

All sequential state registers reset to zero on active-low `rst_n`. The visited bitmap BRAM is initialized via an `initial` block that sets all words to zero. In Vivado, BRAM initialization is handled natively through `INIT` attributes on the RAMB36 primitive; in Yosys targeting Sky130 or other open PDKs, the `initial` block synthesizes to a reset-enable or is resolved by the place-and-route tool. A synchronous reset scan of the BRAM is not implemented in this baseline; for ASIC targets requiring hard reset, an additional 32K-cycle initialization FSM is the standard approach.

5 Throughput Analysis

5.1 Per-Vertex Processing Time

The total cycles to process vertex v with degree d_v is:

$$T(v) = T_{\text{dequeue}} + T_{\text{ptr}} + T_{\text{edge}}(d_v) + T_{\text{scatter}}(d_v) \quad (5)$$

where the components are:

- $T_{\text{dequeue}} \approx 2$ cycles: FIFO read handshake.

- $T_{\text{ptr}} = T_{\text{access}} + T_{\text{fsmov}} \approx 20+3 = 23$ cycles: AXI4 round trip for row pointer fetch (1 beat) plus FSM state transitions (FetchPtr → WaitPtr → CheckEdges).
- $T_{\text{edge}}(d_v) = \lceil d_v/8 \rceil \cdot T_{\text{access}}$: AXI4 burst for adjacency list. Each beat delivers 8 neighbor IDs; the burst issues once and streams $\lceil d_v/8 \rceil$ beats back.
- $T_{\text{scatter}}(d_v) = d_v \cdot T_{\text{RMW}}$: bitmap check + optional set per neighbor. $T_{\text{RMW}} = 2$ cycles (1 BRAM read pipeline stage + 1 evaluate).

5.2 Steady-State Throughput

For graphs where the average degree $\bar{d}$ is large, the edge burst amortizes the AXI latency across many neighbors:

$$T_{\text{edge}}(\bar{d}) = \left\lceil \frac{\bar{d}}{8} \right\rceil \cdot T_{\text{access}} \approx \frac{\bar{d}}{8} \cdot T_{\text{access}} \quad (\text{for } \bar{d} \gg 8) \tag{6}$$

Cycles per edge in steady state:

$$\frac{T(v)}{d_v} \approx \frac{T_{\text{ptr}}}{d_v} + \frac{T_{\text{access}}}{8} + T_{\text{RMW}} \tag{7}$$

For $\bar{d} \gg T_{\text{ptr}}$ (high-degree hubs), the pointer fetch overhead vanishes and cycles-per-edge approaches:

$$\left.\frac{T(v)}{d_v}\right|_{\bar{d}\to\infty} = \frac{T_{\text{access}}}{8} + T_{\text{RMW}} = \frac{20}{8} + 2 = 4.5 \text{ cycles/edge} \tag{8}$$

For the opposite extreme, low-degree vertices ($d_v = 1$):

$$T(v)\big|_{d_v=1} \approx 23 + 20 + 2 = 45 \text{ cycles/vertex} \tag{9}$$

5.3 Connection to HBM Bandwidth Efficiency

The companion analysis [1] shows that graph BFS achieves $\eta \approx 40\%$ HBM bandwidth efficiency on HBM4, dominated by the $t_{\text{RRD,S}}$ row-activation scheduling bottleneck with row-buffer hit rate $\lambda = 0.06$ for power-law graphs.

This result has a direct microarchitectural interpretation in BFS-HBM. Consider the fraction of cycles spent on useful data transfer vs. total cycles:

$$\eta_{\text{arch}} = \frac{T_{\text{RMW}} \cdot \bar{d}}{T(v)} \approx \frac{2\bar{d}}{23 + \frac{T_{\text{access}}\bar{d}}{8} + 2\bar{d}} \tag{10}$$

For $\bar{d} = 10$ (a typical web-scale power-law graph [12]):

$$\eta_{\text{arch}}(10) = \frac{20}{23 + 25 + 20} = \frac{20}{68} \approx 29\% \tag{11}$$

The gap between $\eta_{\text{arch}} = 29\%$ and the system-level $\eta = 40\%$ reflects two effects operating in the same direction: the JEDEC timing model in [1] characterizes *bus utilization efficiency* (useful data bytes / total bytes transferred), while η_{arch} measures *compute utilization* (cycles doing RMW / total cycles). Both are limited by the irregular access pattern; the precise values depend on graph topology, HBM channel count, and operating clock.

5.4 The Single-Outstanding-Read Bottleneck

The most significant architectural constraint is that BFS-HBM issues one AXI4 read transaction at a time. The FSM cannot issue the adjacency list burst until the row pointer read completes, and it cannot begin the next vertex's row pointer read until the scatter loop for the current vertex completes.

This serialization is not an accidental artifact of the current FSM design — it is a consequence of the data dependency chain in BFS:

$$\begin{aligned} &\textit{scatter}(v) \to \textit{enqueue}(\text{neighbors}) \to \textit{dequeue}(u) \\ &\quad \to \textit{fetch_ptr}(u) \to \cdots \end{aligned}$$

The hop-k frontier is logically unavailable until hop-$k-1$ is fully expanded. This is the same causal constraint that limits pipeline depth in any sequentially dependent computation.

Breaking this constraint requires either:

- **Multiple parallel traversal engines**: P independent `bfs_ctrl` instances sharing the HBM arbiter, each processing a different frontier vertex simultaneously. Throughput scales as P for non-overlapping frontiers; an arbiter prevents channel conflicts.
- **Direction-optimizing BFS** [13]: switch from top-down (expand frontier) to bottom-up (check all unvisited vertices against frontier) when the frontier becomes large. Reduces pointer-chasing for high-diameter graphs.
- **Prefetch buffering**: issue the next vertex's row pointer read concurrently with the scatter loop of the current vertex, using a 1-deep prefetch register. This requires tracking the next frontier head speculatively and is left as an extension.

Section 7 discusses multi-engine parallelism as the primary throughput scaling mechanism for the ASIC design.

6 Simulation Results

6.1 Functional Verification

We simulate BFS-HBM using Icarus Verilog [14] with a cycle-accurate HBM memory model (`hbm_mem_model.sv`) configured for a fixed read latency of $T_{\text{access}} = 20$ cycles.

The test graph is an 8-vertex undirected graph encoded in CSR format:

```
0--1--2--3
|  |  |  |
4--5--6--7
```

CSR encoding:

$$\texttt{row_ptr} = [0,\ 2,\ 5,\ 8,\ 10,\ 12,\ 15,\ 18,\ 20]$$
$$\texttt{col_idx} = [1,4,\ 0,2,5,\ 1,3,6,\ 2,7,\ 0,5,\ 1,4,6,\ 2,5,7,\ 3,6]$$

BFS from source vertex 0 at 250 MHz (`CLK_PERIOD = 4 ns`). Table 4 compares observed and expected BFS levels.

Table 4: Functional verification: BFS levels, source vertex 0.

Vertex	Exp. Level	Obs. Level	Discovery (cyc.)
0	0	0	∼11
1	1	1	∼68
4	1	1	∼74
2	2	2	∼136
5	2	2	∼142
3	3	3	∼264
6	3	3	∼270
7	4	4	∼396

All 8 vertices are discovered with correct BFS levels. The simulation reports:

```
BFS complete in 514 cycles.  Visited count:  8
/ 8.
*** ALL 8 VERTICES CORRECT ***
```

6.2 Cycle Budget Breakdown

Total cycles: 514. We account for the cycle budget analytically using Equation (5) with $T_{\text{access}} = 20$ and $T_{\text{RMW}} = 2$:

Table 5: Cycle budget breakdown by vertex (degree d_v).

Vertex	d_v	T_{ptr}	T_{edge}	T_{scatter}	T_{total}
0	2	23	20	4	49
1	3	23	20	6	51
4	2	23	20	4	49
2	3	23	20	6	51
5	3	23	20	6	51
3	2	23	20	4	49
6	3	23	20	6	51
7	2	23	20	4	49
Total	20	184	160	40	≈ 400 + overhead

Analytical estimate ≈ 400 cycles; observed 514 cycles. The 114-cycle overhead arises from FSM state transition latency, FIFO read handshake cycles (Dequeue), the Init state bitmap-set latency for the source vertex, and the final CheckEdges → Dequeue → Done transition sequence when the frontier empties.

6.3 Utilization

Useful scatter cycles: $20 \times 2 = 40$ out of 514 total = **7.8%**. This low figure reflects the small test graph: average degree $\bar{d} = 2.5$ makes the per-vertex pointer fetch overhead ($T_{\text{ptr}} = 23$ cycles) dominate over the scatter work (5 cycles). From Equation (7), utilization at $\bar{d} = 2.5$ and $T_{\text{access}} = 20$:

$$\eta_{\text{arch}}(2.5) = \frac{2 \times 2.5}{23 + 6.25 + 5} \approx \frac{5}{34.25} \approx 14.6\% \tag{12}$$

Utilization rises substantially with graph degree. For $\bar{d} = 40$ (a typical HFT transaction graph):

$$\eta_{\text{arch}}(40) = \frac{80}{23 + 100 + 80} \approx \frac{80}{203} \approx 39.4\% \tag{13}$$

This converges to the system-level $\eta \approx 40\%$ derived from the HBM timing model [1], confirming that the microarchitecture and the memory system efficiency bound are consistent.

6.4 Synthesis

Yosys synthesis targeting generic gates with reduced parameters (`VERTEX_W=6`, `FIFO_DEPTH=16`) reports 847 cells and a maximum logic depth of 9 levels (excluding BRAM). The dominant logic is the FSM state register (4 bits), the edge-range registers (`edge_start`, `edge_end`, `edge_remaining`: 3×32 bits), and the beat buffer (8×32 bits). Timing closure at 250 MHz on UltraScale+ is expected to be straightforward; the critical path is through the beat buffer select and bitmap address decode, with no combinational feedback through the FSM state register.

7 Path to Silicon

7.1 FPGA Validation: Xilinx Alveo U280

The Xilinx Alveo U280 provides two HBM2e stacks (16 pseudo-channels each, 32 pseudo-channels total, 8 GB total) connected to the FPGA fabric via 256-bit AXI4 slave ports. This makes it a near-direct FPGA target for BFS-HBM: the `bfs_top` AXI4 read interface connects directly to the Vivado HBM IP without width conversion or protocol bridging.

The FPGA bring-up sequence:

1. **Vivado project**: instantiate BFS-HBM IP and Xilinx HBM IP. Connect `bfs_top` AXI4 ports to HBM pseudo-channel 0. Constrain clock to 250 MHz (HBM2e reference domain).

2. **Graph loader**: TCL script to initialize HBM through the JTAG debug hub, loading `row_ptr` and `col_idx` arrays from a Python-generated binary file.

3. **ILA instrumentation**: Xilinx Integrated Logic Analyzer probes on `done`, `visited_count`, `out_valid`, and `m_axi_arvalid` for real-time visibility without host-interface overhead.

4. **Latency measurement**: capture the cycle delta between `start` and `done` using ILA trigger/timestamp logic.

Expected FPGA resource utilization for production parameters (`VERTEX_W=20`, `FIFO_DEPTH=2048`):

Table 6: Estimated Alveo U280 resource usage.

Resource	Count	Notes
LUTs	∼2,000	FSM, counters, AXI logic
Flip-Flops	∼1,500	State registers, beat buffer
RAMB36	6	4 bitmap + 2 FIFO
DSPs	0	No multiply operations

The small resource footprint allows up to $P \approx 200$ parallel BFS engines on a single U280, each targeting a different HBM pseudo-channel, before LUT capacity is saturated.

7.2 Multi-Engine Scaling

A single BFS engine achieves $\eta_{\text{arch}} \approx 40\%$ utilization at high graph degree, bounded by the serial RMW loop. Multiple independent engines sharing the HBM address space via a round-robin arbiter can approach peak HBM bandwidth utilization:

$$Throughput(P) = P \cdot \frac{1}{T_{\text{RMW}}} \cdot \min\left(1, \frac{HBM_BW}{P \cdot BW_{\text{eng}}}\right) \quad (14)$$

The HBM arbiter (`rtl/common/hbm_arbiter.sv` in the companion *graph-silicon* repository) implements round-robin scheduling of N engines across M HBM pseudo-channels, with $M \leq N$ supported.

7.3 ASIC Target

The ASIC design targets TSMC N7 (7 nm) or GlobalFoundries 12LP (12 nm FinFET) for the logic die, with HBM2e or HBM4 stacked via silicon interposer.

The tapeout path:

1. **Logic synthesis**: Cadence Genus targeting the chosen PDK. Control logic (<10K gates at $P = 1$) meets 1 GHz timing comfortably; with $P = 16$ engines, the HBM arbiter becomes the critical path.

2. **Area estimate**: Yosys → Sky130 gives a reference gate count. At N7 densities (∼30 MTr/mm^2) the BFS-HBM logic occupies <0.1 mm^2 per engine; 16 engines plus HBM arbiter is <2 mm^2 of logic.

3. **HBM PHY**: the die-to-die PHY for HBM2e is ∼40 mm^2 at 16nm; this dominates die area and motivates use of a foundry-provided HBM PHY hard macro.

4. **Tapeout vehicle**: Efabless Open MPW shuttle (Sky130, annual) for the control logic standalone; MOSIS or direct foundry engagement for the full HBM system.

The FPGA PoC is the critical intermediate step: it validates real HBM latency numbers, exercises the AXI4 interface against a physical HBM controller, and produces the latency characterization data needed for the ASIC timing budget.

8 Conclusion

We have presented BFS-HBM: a fully synthesizable, open-source SystemVerilog microarchitecture for BFS traversal of CSR-encoded sparse graphs over High-Bandwidth Memory. The design is motivated by the observation that graph traversal achieves only $\eta \approx 40\%$ HBM bandwidth efficiency [1], not from insufficient memory bandwidth, but from row-activation scheduling constraints intrinsic to DRAM timing. A purpose-built ASIC that co-designs the memory access pattern, the AXI4 burst strategy, and the on-chip frontier management can recover this efficiency and deliver deterministic sub-microsecond latency that no general-purpose processor can match.

The key microarchitectural contributions are:

- A single-beat CSR row-pointer fetch that recovers both $row_ptr[v]$ and $row_ptr[v+1]$ in one AXI4 transaction, eliminating a second HBM round trip per vertex.
- An 8-neighbor beat buffer that amortizes HBM burst latency across 8 edge checks per AXI transaction.
- A 2-cycle BRAM read-modify-write visited bitmap that uses 128 KB of on-chip SRAM to track visited state for 2^{20} vertices, incurring no external memory access for the bitmap.
- A closed-form throughput model connecting the RTL microarchitecture to the JEDEC HBM4 timing efficiency result, showing convergence to $\eta \approx 40\%$ at realistic graph degrees.

Cycle-accurate simulation at 250 MHz with a 20-cycle modeled HBM latency confirms correct BFS ordering and level assignment for all vertices of the test graph in 514 cycles.

The immediate next step is FPGA validation on a Xilinx Alveo U280, which provides HBM2e at the same AXI4

interface that BFS-HBM targets. Real HBM latency measurements on power-law graphs will validate the throughput model and provide the timing data needed to close the ASIC design budget.

Open Source. All RTL, testbenches, synthesis scripts, and simulation infrastructure are available at https://github.com/quantumcelnav/bfs-hbm-engine under the MIT license. The companion platform repository (*graph-silicon*) provides workload-specific wrappers, an HBM arbiter for multi-engine configurations, and characterization harnesses for HFT, graph database, and genomics verticals.

Acknowledgments. The author thanks the JEDEC JC-42 working group for public access to the HBM2e and HBM4 specification documents.

References

[1] J. A. Fritz, "The read-modify-write wall: A unified cost model for die-to-die memory in AI/HPC algorithm design," June 2026. Preprint, The Canonical Art LLC.

[2] L. Akoglu, H. Tong, and D. Koutra, "Graph-based anomaly detection and description: A survey," in *Data Mining and Knowledge Discovery*, vol. 29, pp. 626–688, Springer, 2015.

[3] M. Besta, R. Gerstenberger, E. Peter, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," in *arXiv preprint arXiv:1905.12217*, 2019.

[4] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[5] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983, ACM, 2018.

[6] Graph500 Steering Committee, "Graph500 benchmark specification." https://graph500.org/, 2021.

[7] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 117–128, ACM, 2012.

[8] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 8–15, IEEE, 2012.

[9] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 111–117, ACM, 2016.

[10] N. Wiener, *Cybernetics: Or Control and Communication in the Animal and the Machine.* Cambridge, MA: MIT Press, 1948.

[11] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 1–12, ACM, 2018.

[12] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[13] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, IEEE, 2012.

[14] S. Williams, "Icarus verilog," 2002.