2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

High-Bandwidth Memory (HBM) in Custom Compute Systems: An Architectural Exploration for Future Computing Paradigms

Puneet Kaushik
Synopsys, India
email id: pkaushik@synopsys.com

Ratnala Naga Sai Mani Krishna Madana
Synopsys, India
email id: rmadana@synopsys.com

Vishal Kumar
Synopsys, India
email id: vishalku@synopsys.com

***Abstract*-High-Bandwidth Memory (HBM) is essential for next-generation High-Performance Computing (HPC) systems, but integrating it into custom compute systems requires careful architectural exploration to optimize performance, power efficiency, and cost. This paper presents a SystemC-based Transaction Level Modeling (TLM) methodology for exploring custom HBM organization with custom compute architectures, enabling a 6-8 month left shift in design timelines. Using HBM4 as a baseline, we demonstrate how architectural exploration can achieve greater memory capacity, lower interface power, and silicon area savings. By addressing challenges in memory allocation, transaction scheduling, design trade-offs, and power optimization for custom compute units, our approach ensures efficient and scalable custom HBM designs, critical for meeting the demands of modern computing workloads. We were able to achieve a data rate of up to 10 gigabits per second (Gbps) and a bandwidth of up to 2.56 terabytes per second (TB/s) per stack.**

I. Introduction

High-Bandwidth Memory (HBM) has become a cornerstone technology for next-generation computing systems, particularly for High-Performance Computing (HPC) and AI/ML workloads. Its ability to deliver unparalleled memory bandwidth and energy efficiency makes it indispensable for data-intensive applications. However, integrating HBM into custom computing systems presents significant challenges, including optimizing performance, power efficiency, and total cost of ownership (TCO). These challenges necessitate a systematic approach to architectural exploration, enabling designers to balance competing constraints and unlock the full potential of HBM. The latest iteration, HBM4, represents a significant leap forward in memory technology. With a data rate of up to 10 gigabits per second (Gbps) and a bandwidth of up to 2.56 terabytes per second (TB/s) per stack, HBM4 is designed to meet the demands of AI accelerators, graphics, and HPC applications [1][2][3]. Its features include support for up to 16 stacks, 32 pseudo-channels, and channel densities of up to 32 Gb [2][3]. These advancements enable higher memory density, faster data handling, and the ability to support advanced multi-tasking, making HBM4 ideal for applications such as AI training and inference, cloud computing, edge devices, and GPUs.

In the current industry landscape, the demand for custom computing systems is rapidly increasing, driven by the need for domain-specific architectures tailored to unique workloads. Hyperscalers, cloud providers, and AI hardware developers are increasingly adopting custom silicon to optimize performance [4], power efficiency, and TCO. However, integrating HBM4 into these custom systems is not straightforward. The complexity of modern workloads, such as AI training and inference, requires careful optimization of memory allocation, transaction

scheduling, and power consumption. Without systematic architectural exploration, designers risk underutilizing HBM's capabilities, leading to suboptimal performance, higher costs, and longer development cycles.

Industry innovations, such as Marvell's custom HBM architecture [5], highlight the importance of optimizing memory capacity, power, and area (Fig. 1). However, gaps remain in the holistic exploration of custom components, advanced simulation tools, and the integration of emerging technologies like HBM.

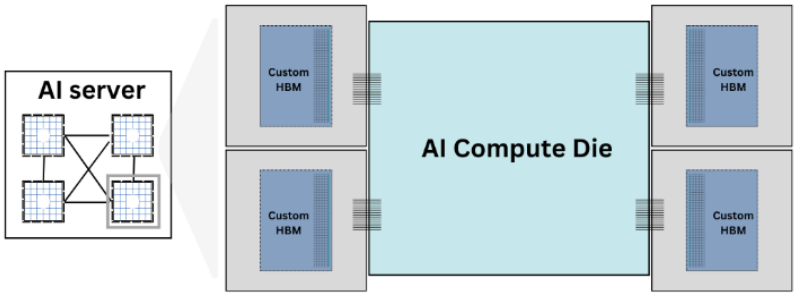

Fig. 1, Custom HBM integrated with AI compute die

In this paper, we first review related work on early performance analysis and architecture exploration. We then introduce our methodology for developing generic TLM-based models for Processing Elements (PEs), Memory Controllers, and System Interconnects. This approach facilitates design space exploration across components to achieve desired performance metrics in terms of throughput, latency, power, and area. Our methodology demonstrates how one can effectively enable customization across all compute components for specific applications and use cases. The feasibility of this approach is demonstrated by the results.

II. Related Work

The integration of High-Bandwidth Memory (HBM) into custom computing systems has been extensively studied to meet the increasing demands of High-Performance Computing (HPC) and AI/ML workloads. Early research demonstrated significant improvements in bandwidth and energy efficiency when utilizing HBM in GPUs and HPC systems [6][11][15]. Custom memory architectures, exemplified by TPUs and deep learning accelerators, highlight the advantages of tailoring memory hierarchies to specific workloads [12]. Heterogeneous systems that incorporate HBM have shown improved data movement efficiency [6][11].

Our approach introduces a methodology for rapid prototyping and optimization of custom components in heterogeneous architecture designs, addressing gaps in architectural exploration of custom HBM compute systems. Utilizing custom interfaces at the design component level, this methodology explores the entire organization at the SoC level [8][9][12]. Additionally, an analysis engine is provided, offering various analysis charts and traces across design components to enable effective root cause analysis for architectural exploration.

III. Methodology

This section describes the methodology (Fig. 2) and analysis of SoC architecture. The hardware platform simulates the processor, interconnect, and memory resources of the intended SoC. The workload model incorporates task-level interdependence, parallelism, and the processing and communication needs for each job, regardless of the architecture. Virtual Processing Units (VPUs) mimic resources like CPUs, GPUs, DSPs, hardware accelerators, or DMAs [15] to map task-based application workload models. Transaction-level modeling (TLM) of the shared interconnect and memory subsystems is driven by the VPUs' conversion of the task graph's communication needs into transactions [8][9]. This enables precise modeling of on-chip bus protocols, replicating dynamic effects such as bus arbitration, DDR latencies, and Quality-of-Service (QoS) regulators. This workload-based virtual prototyping aligns with macro-architecture definition requirements, is independent of specific hardware or software

implementations, and supports HW/SW partitioning by exploring task mapping to various processing resources [12][13].

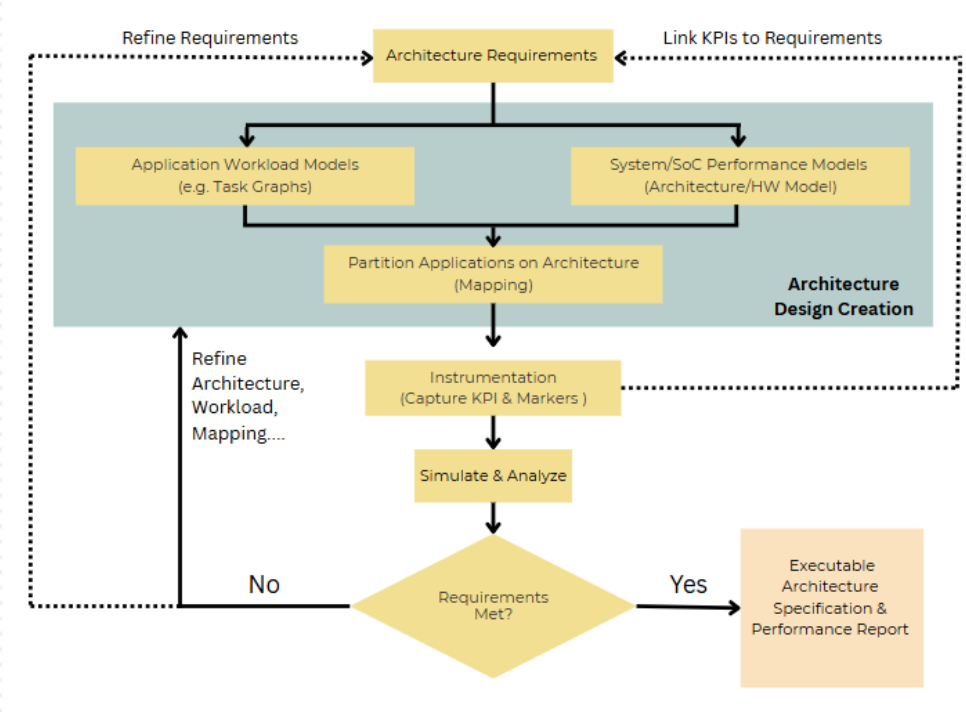

Fig. 2, Approach of Architecture Exploration

The framework provides a requirement and KPI (Key Performance Indicator) domain, which allows us to capture requirements, link them to sub-requirements (e.g., read/write throughput, page hit count, etc.), and roll out the results for individual requirements. To analyze which configuration will meet the requirements defined in terms of KPIs, we use scenario sweeping [13]. Scenario sweeping is a structured way to explore the design space across the architecture design, configure parameter values of various components of the design, configure workload flow mappings, configure other TLM simulation settings (including analysis), and measure the influence of the configuration changes on the results. Each individual design configuration is called a scenario.

Every new scenario contains the default configuration of the design created. Multiple scenarios can be added and configured. We specify a range or set of values to be used for each sweep parameter, and the framework generates scenarios by using all the combinations of specified configurations across the design components such as DSP, NoC, DRAM, etc. After successful scenario simulation run completion, the analysis information enabled gets processed and generates analysis results. These results are then displayed in the results table for a comprehensive study of analysis. The design can be analyzed based on these results, and we set and redefine criteria for further studies on performance optimization and architecture exploration.

A. Generic Mutli Ported Memory Controller (GMPMC)

To explore the integration of HBM into custom compute systems, we utilized a SystemC-based Generic Multi-Ported Memory Controller (GMPMC) [7][6][15]. The GMPMC is a configurable memory controller model (Fig. 3) that can be used to simulate different supported memory types. The model provides high time fidelity on the SDRAM access memory path, simulating the latency effects of the application/workload. It deals with both static and dynamic latencies of HBM's memory access pattern and supports data-accurate transactions for all types of

traffic [1][2][3]. It can be utilized to meet defined KPIs like throughput, latency, utilization, efficiency, etc., for multi-die systems [11]. In our Generic Multi-Ported Memory Controller, to explore architectural use cases, a customized scheduler and arbiter can be put in place to enhance system performance and resource distribution.

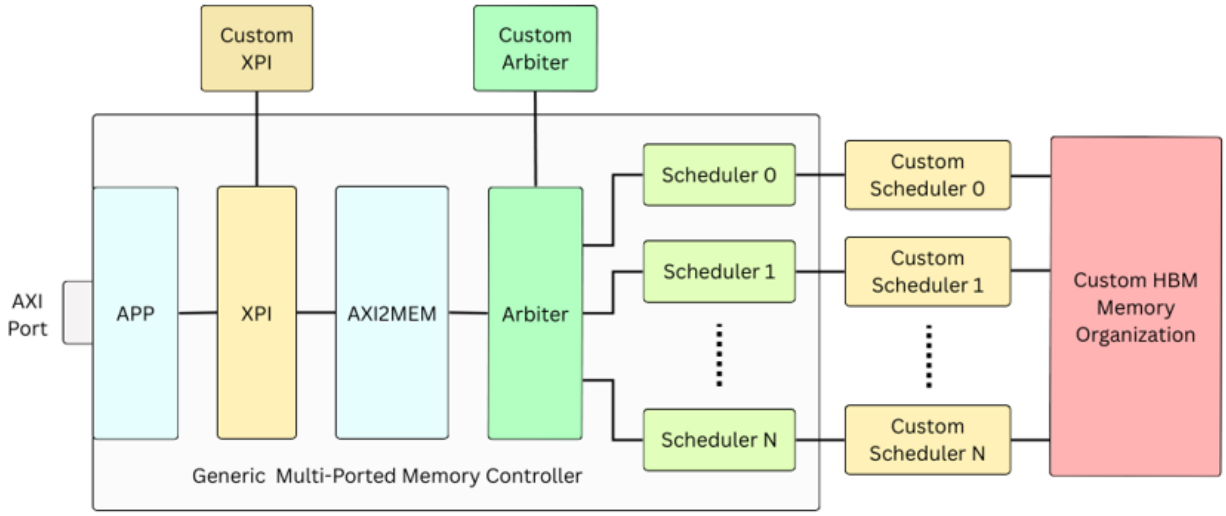

Fig. 3, Generic Mutli-Ported Memory Controller Architecture

B. Virtual Processing Unit (VPU)

The Virtual Processing Unit (VPU), as shown in Fig. 4 is a configurable generic block that has a task manager and a scheduler [12]. It controls the set of tasks that are mapped to it. By default, the VPU contains the memory driver, which is responsible for executing the deterministic read/write access, and a processing element, which executes the compute load of the task.

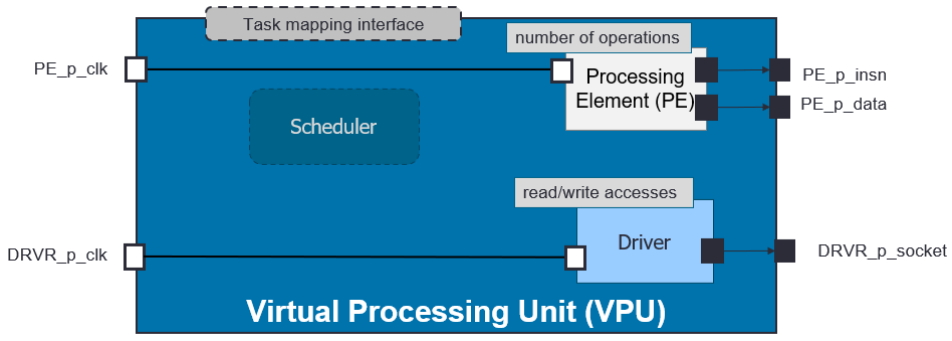

Fig. 4, Block Diagram of Virtual Processing Unit

C. Generic Network on Chip (GenericNoC)

GenericNoC, as shown in Fig. 5, is a configurable interconnect that can be used to connect TLM2.0-based initiators and targets [13]. The bus comes with configuration parameters that are used to specify the desired protocol for the interconnect. Depending on the use case, the topology can be configured as mesh, ring, or custom topology. Apart from topology, Generic NoC has multiple arbitration schemes like non-deterministic (first come, first served basis), round robin (the arbiter remembers the initiator to which the last grant is given), and custom (a user-defined arbiter block is used).

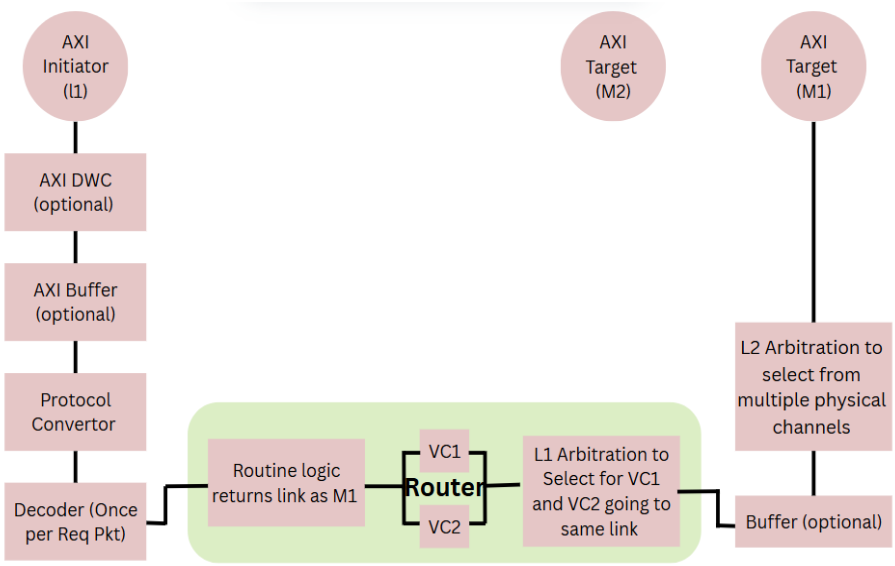

Fig. 5, Transaction Flow in GenericNoC

IV. Design Space Exploration of Custom Compute System

A. *Use Case Description*

We have designed a platform for a high-performance computing (HPC) system based on heterogeneous architecture where we integrate multiple custom compute PEs targeting data-intensive LLM workloads with large address space where diverse access patterns must be efficiently handled along with complex memory hierarchies while minimizing overheads and maximizing performance using architectural exploration.

B. *Platform Description*

This section describes a platform (Fig. 6) containing a Generic Multi-Ported Memory Controller for HBM configuration in custom compute systems [1][3].

- There are four VPUs that serve as traffic initiators, each mapped to a distinct workload, and one VPU is operating as a Register Initiator. The registers of the GMPMC can be configured by mapping a workload to the Register Initiator.
- The data width configuration of both the VPUs and BUS matches the port data width of the Generic Multi-Ported Memory Controller.
- The clock is configured based on the selected speed bin and frequency ratio.
- The configuration of the custom HBM block, which is configured from GMPMC for performance validation, acts as a subordinate.

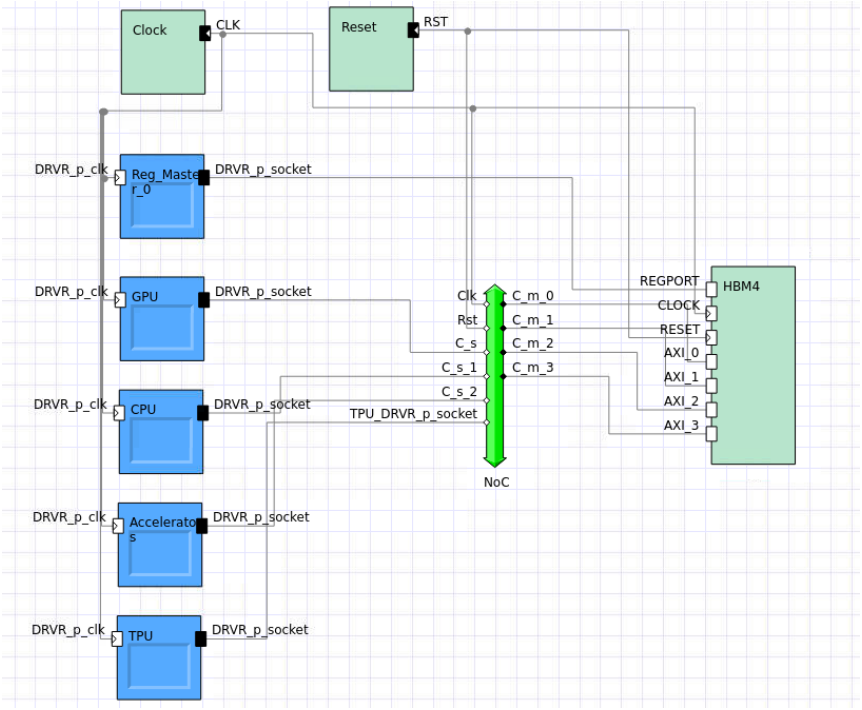

Fig. 6, Custom HBM for Custom Compute Systems

C. Key Performance Indicators (KPIs)

There are various KPIs [13] measures that are used for exploration of custom HBM for custom compute systems to optimize latency and bandwidth, which can be viewed through an analysis framework. There are different reasons why one could be getting low KPI numbers for a particular use case. To identify the performance bottlenecks, analysis views are used. The following are some common KPI metrics and their analysis views available for measuring and improving performance.

Latency: This metric provides a comprehensive view of the delays experienced across different system blocks. As shown in Fig. 7, latency is defined as the total round-trip time taken for a transaction – from the moment it starts to when it completes at a particular block. This latency breakdown helps identify which blocks contribute the most to overall delays [11][15].

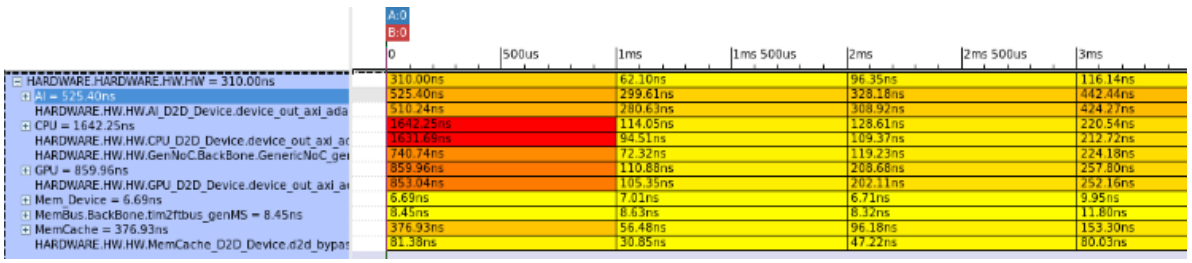

Fig. 7, Latency Analysis View

Utilization: It refers to how much of a time interval a data transfer channel is actively used. From the data shown in Fig. 8, one can evaluate both the command and memory channel utilizations to understand resource usage efficiency. High utilization may indicate efficient resource usage, if it approaches 100%, it could signal potential congestion or saturation, limiting headroom for additional workload.

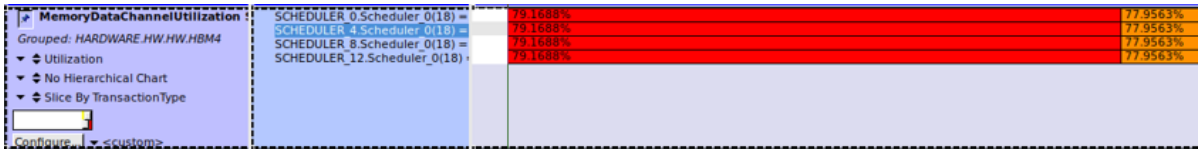

Fig. 8, Utilization Analysis View

Throughput: It is defined as the number of bytes per second for transactions and/or data transfers. An analysis view for throughput is shown in Fig. 9.

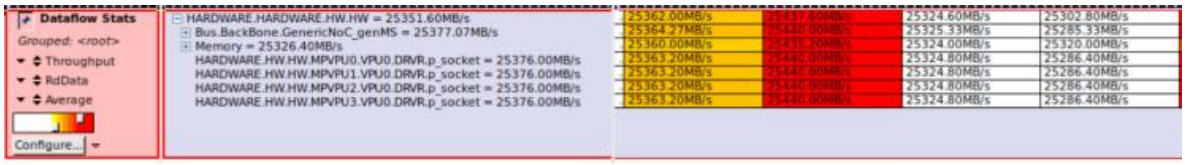

Fig. 9, Throughput Analysis View

CAM Utilization: In High-Bandwidth Memory (HBM) systems, a Content Addressable Memory (CAM) acts as a scheduler that temporarily holds memory transactions before they're sent to the memory interface. Tuning this metric helps in identifying the ideal CAM size for balancing memory access latency and throughput [15].

XPI Buffer Utilization: Each AXI [14] Port Interface (XPI) in the generic model is equipped with dedicated buffers that temporarily store AXI transfers before passing them to the next component. It is valuable for identifying transfer-type-specific bottlenecks and helps detect contention or queuing delays in a particular channel.

Hot Bit Analysis: Hot-bit analysis tracks the frequency of bit transitions in AXI addresses. This is crucial for optimizing the Address Mapping Registers (ADDRMAPx). This enables the generation of accurate hot-bit metrics [15]. High transition activity in certain address bits (hot bits) suggests poor address locality or mapping inefficiencies.

D. Workload Description

We have used 4 workload models, each mapped to the GPU, CPU, Accelerator, and TPU hardware models, respectively. For workload modeling, we use task graphs [10][11].

GPU Workload: In GPU workload, the benchmark typically involves multiple threads or processes executing memory access operations in parallel. These memory access patterns often include scenarios such as sequential access, random access, and patterns that exhibit varying degrees of spatial and temporal locality.

CPU Workload: CPU workload is used in multi-chase benchmarks involving multiple threads or processes executing memory access operations simultaneously.

Accelerator Workload: The accelerator workload provides detailed information about the operations involving large datasets along with their characterization. The workload executes the stream of operations. Each type of operation is depicted with a pipeline model to achieve fine granularity.

TPU Workload: The TPU workload is optimized for large-scale tensor operations in deep learning, excelling at training/inference of neural networks. This delivers high throughput for batch processing via systolic array architecture, suited for TensorFlow/JAX workloads. Unlike GPUs, TPUs prioritize efficiency in matrix math but are less flexible in non-AI tasks.

The following Fig. 10 illustrates the access patterns across processing element dies.

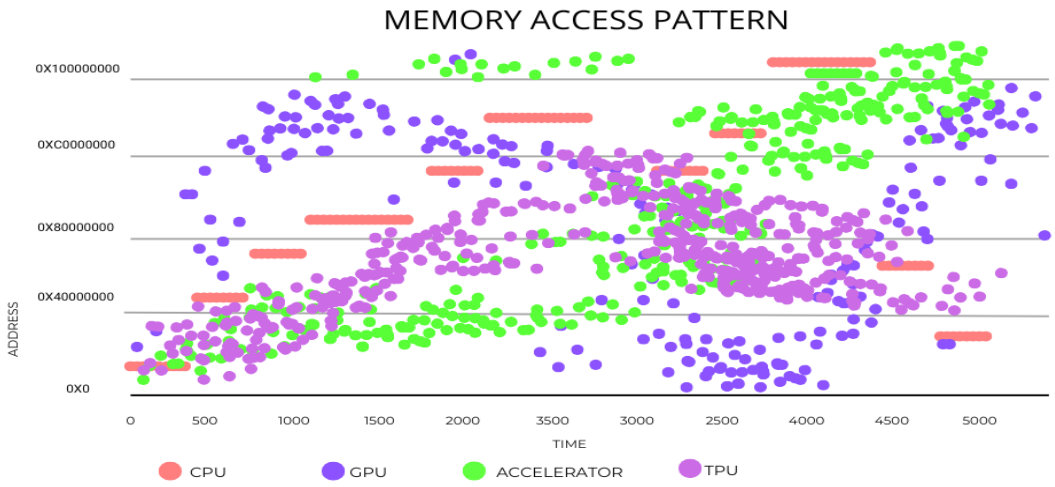

Fig. 10, Access patterns across processing element

E. Optimization of custom HBM and SoC components

Customizing the components of High-Bandwidth Memory (HBM) and System-on-Chip (SoC) is essential for achieving peak throughput and bandwidth tailored to specific workloads [3][6][11][15]. The proposed SystemC-based Transaction Level Modeling (TLM) methodology provides a systematic framework for exploring design trade-offs, enabling users to evaluate and optimize key components such as the arbiter, scheduler, and memory organization [8][9][12]. By simulating workload-specific scenarios, this methodology helps identify bottlenecks, analyze performance metrics, and guide the customization of these components. It empowers designers to make informed decisions by providing insights into how different configurations impact memory allocation, transaction scheduling, and overall system efficiency, ultimately leading to optimized and scalable designs.

Custom Arbiter Design: The arbiter plays a pivotal role in managing access to HBM resources by resolving conflicts between multiple compute units. The methodology allows users to explore various arbitration strategies, such as round-robin, weighted priority, and workload-adaptive policies. Round-robin arbitration ensures fairness by servicing requests in a cyclic order, while weighted priority arbitration assigns different priorities to requests based on workload requirements. Workload-adaptive policies dynamically adjust priorities based on real-time access patterns, ensuring optimal performance for diverse workloads. By modeling and analyzing these strategies, designers can select the most suitable arbitration logic to balance fairness, latency, and throughput. A pseudocode snippet is shown in Fig. 11.

Custom Scheduler: The scheduler determines the order and timing of memory transactions, directly impacting the efficiency of HBM utilization. Customizing the scheduler to align with workload-specific access patterns enables better exploitation of HBM's parallelism. For instance, workloads with high spatial locality can leverage burst scheduling to maximize data transfer efficiency, while workloads with high temporal locality benefit from prefetching strategies. Additionally, incorporating bank-level parallelism into the scheduler design reduces

contention and ensures balanced utilization of HBM banks, further enhancing throughput. A pseudocode snippet is shown in Fig. 11.

Optimized Memory Organization: The organization of HBM, including the arrangement of banks, channels, and stacks, significantly influences performance [2][3]. Customizing the memory organization to match the workload's data access patterns ensures efficient utilization of available bandwidth.

```
Function rd_arbitrate_method()

    Initialize:
        granted_trans ← NULL
        is_granted_hi_priority ← FALSE
        granted_time, last_win_time ← current_time
        granted_port ← INVALID

    For each port_idx in NUM_AXI_PORTS:
        If no request OR just arrived this cycle: continue

        If high-priority already selected AND current is low: continue

        If current is high-priority AND selected is low:
            select current port
            continue

        If current arrived earlier than selected:
            select current port
            continue

        If current was granted less recently:
            select current port
            continue

    Assert granted_trans is not NULL

    Update:
        Decrement pending count
        Clear selected request
        Update last win time
        Mark channel busy

    Set protocol state to ARVALID
    Send transaction via master adaptor

End Function
```

```
Function select_nxt_rd_entry_4_scheduling(rank, bank) → ntt_entry

    For idx from 0 to cam_depth - 1 do

        If rd_cam[idx] is not valid:
            Continue

        If rd_cam[idx].rank ≠ rank OR rd_cam[idx].bank ≠ bank:
            Continue

        is_pg_hit ← (rd_cam[idx].row == open_row_4_bank[rank * num_of_banks + bank])
        is_hi_priority ← (rd_cam[idx].qos ≥ rd_hi_priority_qos_level)

        If is_selected_hi_priority AND NOT is_hi_priority:
            Continue

        If NOT is_selected_hi_priority AND is_hi_priority:
            selected_index ← idx
            is_selected_page_hit ← is_pg_hit
            is_selected_hi_priority ← TRUE
            selected_arrival_time ← rd_cam[idx].arrival_time
            Continue

        If is_selected_page_hit AND NOT is_pg_hit:
            Continue

        If NOT is_selected_page_hit AND is_pg_hit:
            selected_index ← idx
            is_selected_page_hit ← TRUE
            selected_arrival_time ← rd_cam[idx].arrival_time
            Continue

        If selected_index == -1 OR rd_cam[idx].arrival_time ≤ selected_arrival_time:
            selected_index ← idx
            selected_arrival_time ← rd_cam[idx].arrival_time

    End For

    If selected_index ≠ -1:
        ntt_entry.cam_index ← selected_index
        ntt_entry.is_entry_valid ← TRUE
        ntt_entry.is_page_hit ← is_selected_page_hit
        ntt_entry.is_hi_priority ← is_selected_hi_priority
        ntt_entry.arrival_time ← rd_cam[selected_index].arrival_time

    Return ntt_entry

End Function
```

Fig. 11, Custom Arbiter and Custom Scheduler pseudocode snippet

F. Transaction Scheduling and Data Flow Optimization

Efficient transaction scheduling and data flow management are key to minimizing latency and maximizing the bandwidth utilization of HBM4-based systems [3][6][11][15]. By intelligently coordinating memory requests and orchestrating data movement, overall system performance can be significantly enhanced. To achieve this, the following strategies are adopted.

- Priority-Based and QoS-Aware Scheduling with Out-of-Order Execution: To ensure low-latency service for time-sensitive and mission-critical operations, priority-based scheduling and QoS-aware mechanisms are employed [3][6][11][15]. These strategies allow critical transactions to be prioritized under mixed workloads. Additionally, out-of-order execution dynamically records memory transactions to optimize pipeline utilization, minimize stalls, and enhance parallelism.
- Efficiency-Oriented Memory Optimization Techniques: Read-write balance optimization and transaction batching techniques are implemented to reduce switching overhead and command queue pressure [3][6][11][15].

V. Analysis Results

To determine the configuration that satisfies the specified requirements-expressed in terms of key performance indicators (KPIs), a scenario sweeping methodology is used. It provides a systematic approach to explore the architectural design space by tweaking parameter values across various components and assessing the impact of these variations on performance outcomes. Each distinct configuration corresponds to a specific scenario simulation, the resulting analysis data is processed to generate performance metrics. The target objective is to achieve a minimum data channel utilization of 95%, however, under the default configuration, the system achieved approximately 76%. The difference in KPIs before and after tweaking the parameters is shown in Fig. 12.

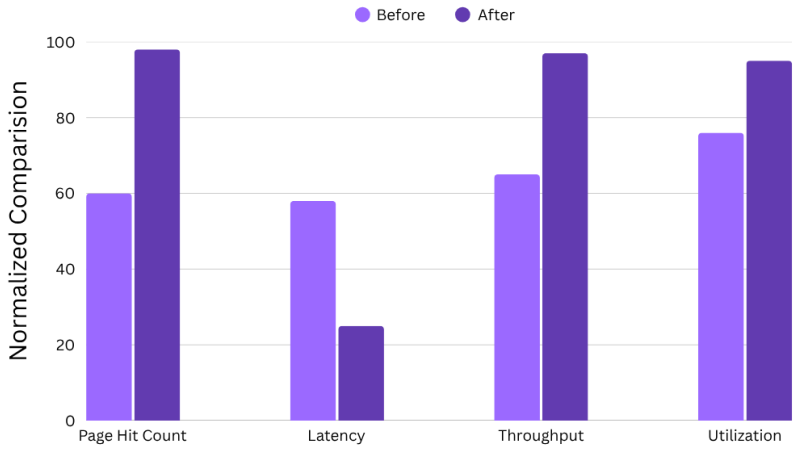

Fig. 12, Normalized Results

To address this performance gap, a platform was developed to simulate the target compute system, and simulations were conducted using incremental read transactions. Transaction-Level Modeling (TLM) port traces [13] were analyzed to gain visibility into memory transfer activity. In this platform, weighted priority-based arbiter and a custom scheduler have been used. As illustrated in Fig. 13, these traces revealed latency gaps in the reception of data, indicating inefficiencies in the scheduling of memory access.

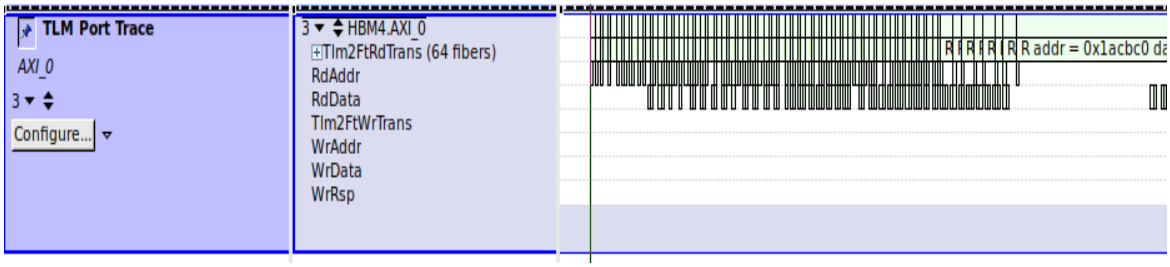

Fig. 13, TLM Port Traces

To further investigate command behavior, Command Channel Utilization traces [13] were examined (Fig. 14). Analysis of both the bank-level and transaction-level views revealed frequent occurrences of simultaneous active and precharge operations directed at the same HBM bank. These conflicts reduced command channel efficiency, highlighting the need for improved scheduling mechanisms designed specifically for HBM-based memory systems in custom compute platforms.

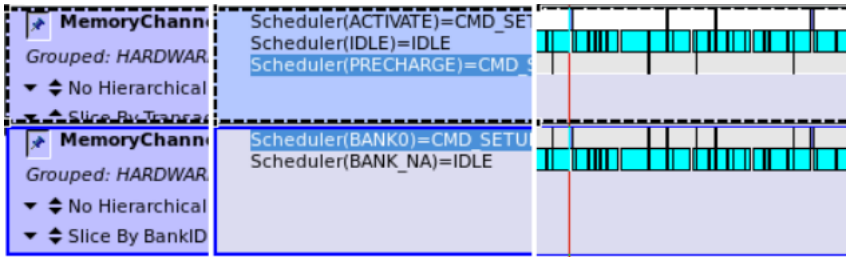

Fig. 14, Command Channel Utilization Traces

Address mapping and hot bit analysis views were used to further find the reason for the unexpected numbers of activates and precharges. Fig. 15 shows the hot bit analysis view of the scenario. The hot bit analysis trace illustrates the frequency with which individual address bits are toggled during memory access. From the trace, it is evident that the row bit exhibits high toggle activity, whereas the bank bit and stack ID bits remain largely inactive. This imbalance indicates that memory traffic is concentrated on the same HBM bank, resulting in repeated page openings and closings, and consequently, an increased rate of page misses. Furthermore, the absence of toggling beyond the 12th bit suggests that bank and stack ID bits are not being utilized, which implies underutilization of the available HBM bandwidth, creating localized contention and performance bottlenecks.

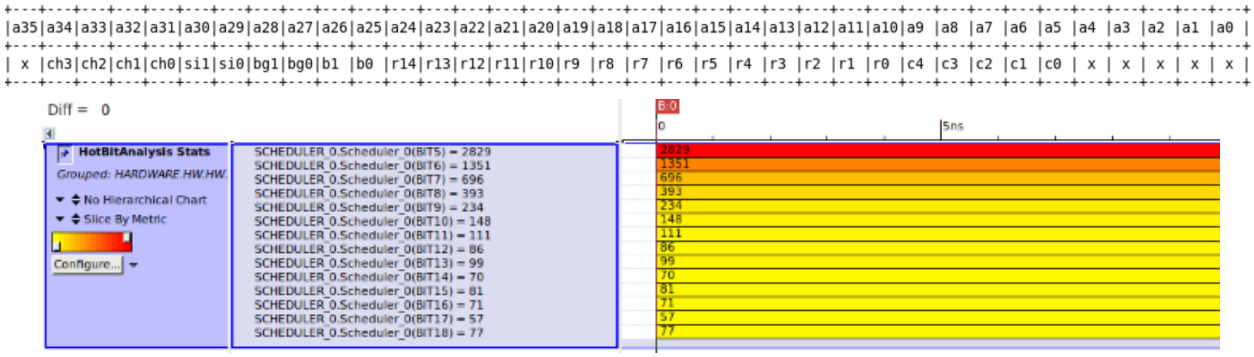

Fig. 15, Address Mapping and Hot Bit Analysis Traces

To address this issue, the address mapping scheme was modified so that the bank and stack ID bits were mapped to the Least Significant Bits (LSBs). This change redistributed memory accesses more evenly across multiple banks within the HBM stack and enabled the design to meet the desired KPI in the custom compute systems.

Prior to optimization, memory channel utilization was measured at approximately 76%, significantly below 95%. Throughput was similarly affected, with observed bandwidth utilization reaching only 68% of the HBM's theoretical peak under read-intensive workloads. To address these inefficiencies, configuration has been changed using analysis view, enabling more uniform traffic distribution across HBM banks and stacks. Post-optimization, channel utilization increased to over 95%, and throughput improved to 97% of the peak bandwidth.

VI. Conclusion

As we know, HBM has become a key component in the design of modern processors and hardware accelerators, offering significantly higher memory bandwidth and lower latency. It plays a pivotal role in enabling compute and bandwidth-intensive applications such as AI training and inference, scientific simulations, real-time data analytics, and high-performance computing (HPC). As these application domains continue to scale in complexity and size, efficient utilization and customization of HBM for compute systems become crucial for meeting performance targets. The proposed methodology demonstrates the potential of leveraging pre-RTL SystemC-based TLM exploration to design custom HBM architectures optimized for AI/ML workloads. By focusing on increasing

memory capacity, optimizing transaction scheduling, and enhancing memory bandwidth utilization, the approach addresses the challenges of integrating HBM into modern compute systems. This systematic exploration enables the development of scalable and efficient architectures tailored to the demands of data-intensive applications, paving the way for advancements in AI accelerators, cloud computing, and edge services.

REFERENCES

[1] Rambus, “HBM4 Controller Product Brief”. (available at: https://go.rambus.com/hbm4-controller-product-brief)

[2] JEDEC Solid State Technology Association, "High Bandwidth Memory DRAM (HBM3)," JESD238B.01, April 2025. (Available at: https://www.jedec.org/standards-documents/docs/ jesd238b01).

[3] JEDEC Solid State Technology Association, "High Bandwidth Memory DRAM (HBM4),".

[4] Marvell, "Marvell Announces Breakthrough Co-Packaged Optics Architecture for Custom AI Accelerators," 2024. (Available at: https://www.marvell.com/company/newsroom/marvell-announces-breakthrough-co-packaged-optics-architecture-for-custom-ai-accelerators.html)

[5] Marvell, "Marvell Announces Breakthrough Custom HBM Compute Architecture to Optimize Cloud AI Accelerators," 2024. (Available at: https://www.marvell.com/company/newsroom/marvell-announces-breakthrough-custom-hbm-compute-architecture-to-optimize-cloud-ai-accelerators.html)

[6] H. Jun et al., "HBM (High Bandwidth Memory) DRAM Technology and Architecture," 2017 IEEE International Memory Workshop (IMW), Monterey, CA, USA, 2017, pp. 1-4, doi: 10.1109/IMW.2017.7939084

[7] IEEE Standard for Standard SystemC® Language Reference Manual by Design Automation Standards Committee of the IEEE Computer

[8] Ghenassia, Frank. “Transaction-level modeling with SystemC”. Springer, 2005.

[9] T. Grotker, S. Liao, G. Martin, S. Swan. ¨ System Design withSystemC. Kluwer Academic Publishers, 2002

[10] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, B. Vanthournout, "A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms", Design,

[11] G. Srivastava, P. Kaur, P. Kaushik and M. Gupta, "Optimizing High Bandwidth Memory in Multi-Die Systems," 2023 IEEE Women in Technology Conference (WINTECHCON), Bangalore, India, 2023, pp. 1-5, doi: 10.1109/WINTECHCON58518.2023.10276441.

[12] A. Dudeja, A. Tara, A. Garg, T. Jain, “Framework for creating performance model of AI algorithms for early architecture exploration”. 2019 https://easychair.org/publications/preprint/JC7M

[13] AMBA AXI Specification by ARM (Available at: https://developer.arm.com/documentation/ihi0022/latest)

[14] Rchard Solomon, “DMA: Its Importance and Applications Explained” (available at: https://www.synopsys.com/blogs/chip-design/dmaenhances-computer-performance.html).

[15] J.-H. Chae, "High-Bandwidth and Energy-Efficient Memory Interfaces for the Data-Centric Era: Recent Advances, Design Challenges, and Future Prospects," Dept. of Electronics and Communications Engineering, Kwangwoon University, Seoul, South Korea.