An Event-Driven Graph Processing Accelerator Implementation based on IBM 130nm Technology

EECS 627: VLSI II Final Project Report

Ganrun Xu, Wenhan Kou, Peter Zhong, Xiuji Wu, Yu-Sheng Ting, Zetong Guan
Electrical & Computer Engineering, University of Michigan
{ganrunxu, kwenhan, hpzhong, xiujiwu, yushengt, ztguan}@umich.edu

***Abstract*—Graph Processing has become a very important computation workload in various domains. As real-world graphs are massive, such workloads are memory bound. The irregularity of the graph datasets exacerbates this problem because the conventional architectures are only optimized for regular memory access patterns. Therefore, Shafiur et al. [1] proposed the Event-Driven Graph Processing Model to convert random memory accesses to sequential ones and exploit the algorithmic insights to coalesce events. They also proposed the first accelerator, GraphPulse, running on this execution model. Although the accelerator has been verified to be efficient on the simulator, it has not been verified from the VLSI implementation level. In this project, we design the microarchitecture in Verilog of GraphPulse for PageRank, verify the functionality with a Python-based cycle-accurate golden brick, and implement it on IBM 130nm technology.**

***Index Terms*—Graph Processing, PageRank, Event-Driven Execution Model**

I. Background & Motivation

GRAPH processing has become a very important workload in various domains, including social networks, recommendations, web graphs, financial fraud detection, and bioinformatics. Graph algorithms such as PageRank, Betweenness Centrality, and Collaborative Filtering have been widely used to extract insights from massive real-world datasets, which normally have billions of vertices and edges.

Computer architects are faced with several challenges to optimize the efficiency of large-scale graph processing. Firstly, graph processing is memory-intensive. The computation workload is normally simple while requiring a great amount of data. Secondly, most popular implementations are based on the bulk-synchronous model, which suffers great synchronization overhead. Thirdly, the overhead to track the active vertices or edges is substantial.

To deal with the challenges above, Shafiur et al. [1] proposed Event-Driven Execution Model and a graph processing accelerator, GraphPulse, based on this model. Although it has been proven to be efficient at the architecture level, whether it is practical and scalable in chip design is not yet evaluated. In our project for EECS 627: VLSI II, we aim to implement a mini-version of GraphPulse accelerator with IBM 130nm technology and analyze the pros and cons of the architecture at the VLSI level.

II. Event-Driven Execution Model

Vertex-centric models can be further divided into Push and Pull styles, each being expensive in part of its memory access pattern, as shown in Fig. 1. In push style, the vertex property modifications are pushed to outgoing neighbors from a currently active vertex. Because multiple vertices can push modifications to a shared neighbor, atomic operations are required to guarantee correct results. In pull style, the vertex property modifications are pulled by a currently active vertex from its incoming neighbors. Because the vertex has no information on which neighbors have valid modifications, it has to pull from every neighbor, resulting in redundant memory reads. In both push and pull styles, due to the irregularity of connections between vertices, random accesses to the vertex property array are inevitable.

The Event-Driven Execution Model defines "events" as the basic unit of scheduling workloads. An event is composed of two fields: destination vertex ID and the delta of the vertex property. When processing an incoming event, Reduce function is called to update the destination vertex property, and then Propagate function is executed to generate a series of new events to update the neighbors. The benefits of the event abstraction are three-fold: Firstly, as each event is only associated with one vertex, there's no need to deal with atomic updates of vertex properties. Secondly, the scheduling of events is isolated from the computation and memory access. Thirdly, no active set maintenance is needed, because the in-flight events intrinsically track the active vertices.

Two major optimizations are therefore made possible. Firstly, because most Reduce functions are commutative and associative, events can be sorted with respect to the destination vertex ID before issuing. The scheduler can therefore issue events with consecutive destination vertex IDs, whose vertex properties are also stored adjacently in the memory. In this way, the random accesses in conventional vertex-centric models are converted to sequential accesses in the Event-Driven model. Secondly, because most Propagate functions are distributive, the newly generated events can coalesce with the not-yet-issued events, so fewer in-flight events need to be stored and assigned to processing elements.

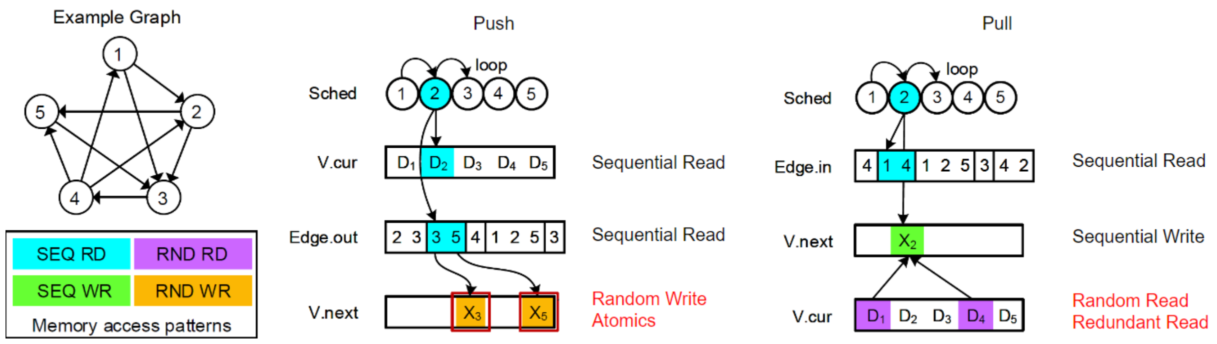


Fig. 1: Memory access pattern of Push and Pull style vertex-centric graph processing model [1]

III. GraphPulse Micro-Architecture

A. Overview

Fig. 2 below shows an overview of the version of the GraphPulse architecture that we have implemented. Our implementation is designed specifically to handle the PageRank algorithm. The two major components in our architecture are event queues and event processors. They are responsible for storing events and handling events respectively. We added a scheduler, an output buffer, and crossbars to support the communication and operation of these two major components. The detailed specifications and functions of each of these modules will be introduced in the subsections below.

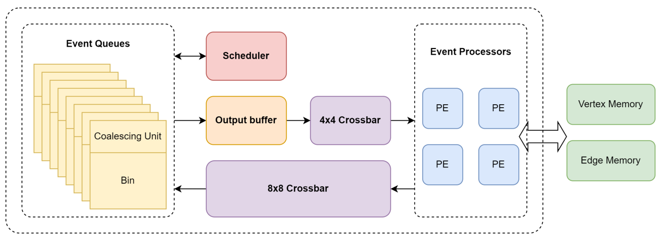


Fig. 2: Project overview

B. Event Queues & In-place Coalescing Unit

Events are directly mapped to a queue of 8 bins, 4 rows, and 8 columns, as shown in Fig. 3. In order to spread the graph clusters over multiple bins, vertices are mapped in a row-bin-column order. One bin is selected to read by a Round Robin arbiter at a time. In the bin, a valid row with the least significant index is chosen by a priority encoder to read.

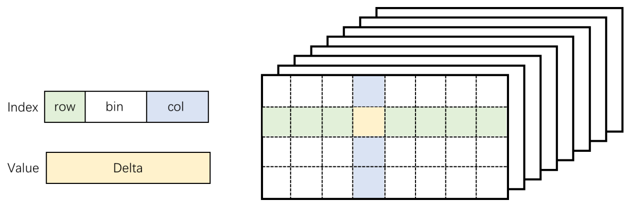


Fig. 3: Event Queues mapping

To combine multiple events of the same destination, we have an in-place coalescing unit for each bin. The pipeline structure is shown in Fig. 4. The coalescing unit is a pipeline structure with an input buffer. Incoming events are stored in the input buffer at first, then issued to the pipeline structure. For an issued event, its index will be used to search for the current value stored in the queues. After adding up the value of the issued event and the search result, the new value will be written back to the event queues.

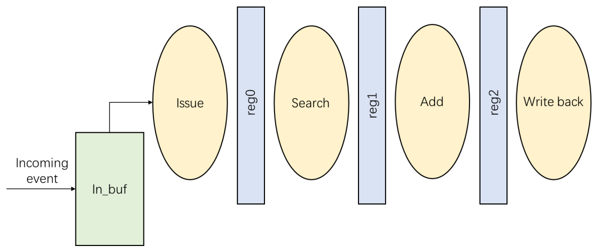


Fig. 4: In-place Coalescing Unit pipeline

C. Event Scheduler

The scheduler is a six-stage finite state machine. The state transitions are illustrated in Fig. 5. "I" is the initialization state. The "C" state is where the round-robin arbiter starts to choose the next reading bin. If the chosen bin is valid, the coalescing unit of it will stop taking new events into registers in the "B" state. If one bin is selected again, the scheduler will enter the "D" state, waiting for all the Event Processors to be idle, then go to the "W" state. Otherwise, it will directly come to the "W" state, waiting for the coalescing unit to be clean. Once the coalescing unit is clean, it will go to the "R" state and set the read enable signal to 1.

D. Crossbar from Event Queues to Event Processors

When a row of events is drained from the Event Queues by the scheduler, they are placed into an output buffer, the structure of which is shown in Fig. 6. The output buffer first filters out the invalid events (or "bubbles"), so these invalid events will not waste resources and time to be routed and assigned to Event Processors. This functionality is implemented as an implicit crossbar. And then the valid events are pushed into a FIFO. The oldest events in FIFO are exposed to the crossbar, which is responsible for assigning events to idle processors.

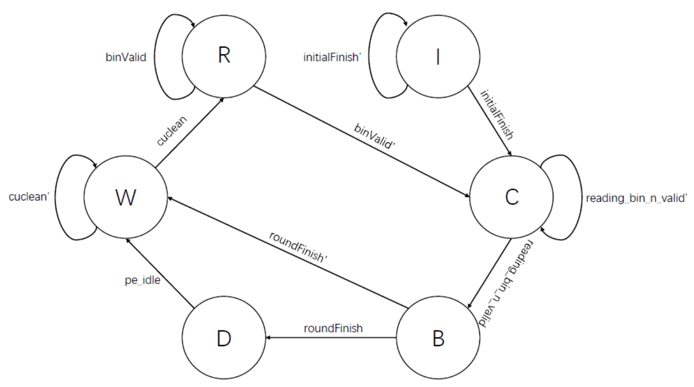
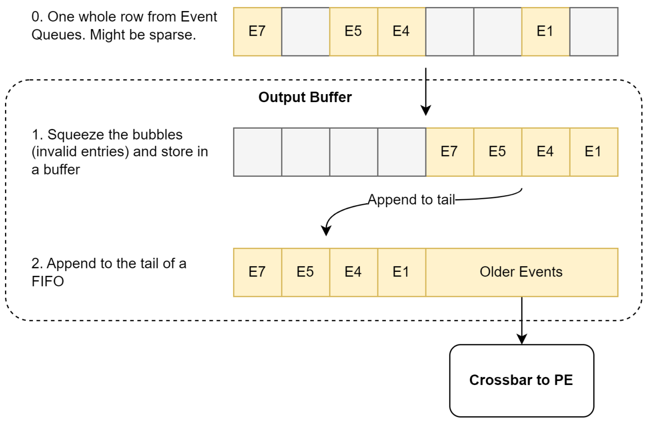


Fig. 5: Event Scheduler FSM

In this way, the oldest events are guaranteed to be assigned to Event Processors first.



0. One whole row from Event Queues. Might be sparse.
E7
E5
E4
E1
Output Buffer
1. Squeeze the bubbles (invalid entries) and store in a buffer
E7
E5
E4
E1
Append to tail
2. Append to the tail of a FIFO
E7
E5
E4
E1
Older Events
Crossbar to PE

Fig. 6: Output Buffer

Next, the events are handed over to the event routing crossbar, as shown in Fig. 7. It maintains a list of idle Event Processors. At the cycle when a processor asserts the “ready” signal, the crossbar gets notified and pushes the PE ID into the list. At the cycle when an event is selected to be assigned to a PE, the PE ID is removed from the list. Because the list is implemented as a FIFO, it guarantees that the oldest idle PEs get assigned an event first. The number of pipeline stages is parameterized in Verilog codes so that the number of cycles through the crossbar can be adjusted according to the system clock period for optimized performance. As we have four PEs in the system, each receiving one event per cycle, a 4x4 crossbar is implemented.

E. Event Processors

The event processors are finite state machines (FSMs) with 4 different states, each corresponding to a stage of operation performed during handling an incoming event from the event queue. These four states are INIT, IDLE, RUW, EVGEN. Each processor also has a 4-stage pipelined 16-bit floating point unit (FPU) along with the FSM structure. The FPUs are capable

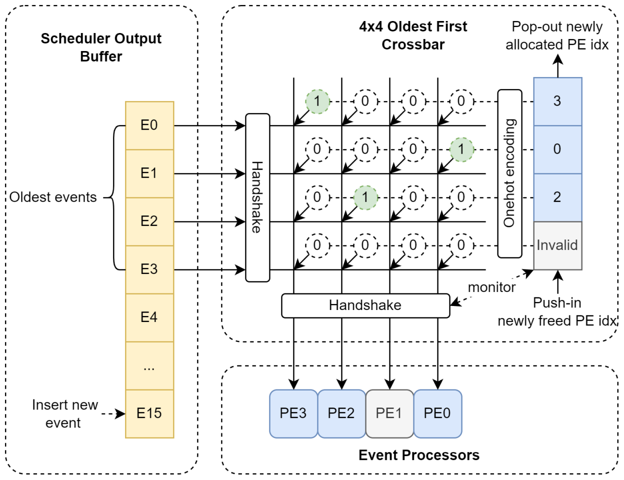
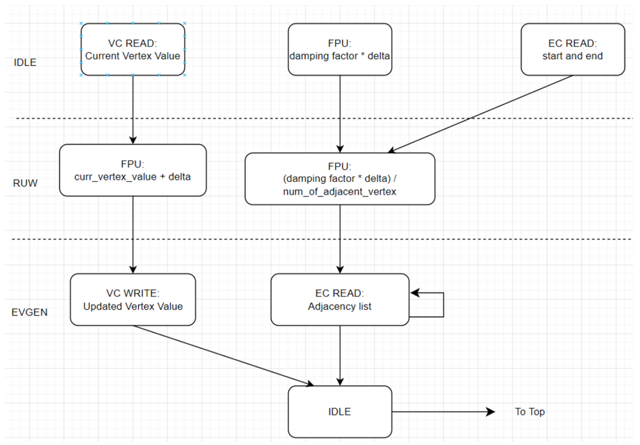


Fig. 7: 4x4 crossbar from Event Queues to Event Processors

of performing half-precision floating point addition, subtraction, multiplication, and division according to the IEEE-754 standard.

Overview of Single Processor Operations: The event processor is initially reset to the INIT state. When reset is deasserted, the processor will generate one initial event per vertex to distribute the initial PageRank score among all of the existing vertices of the graph. These initialization events will be received and stored in the downstream event queue. After successfully generating and sending out these events, the processor will transition to the IDLE state to wait for new events from the event queue. Once a new valid event is present at the input of the processor, the processor will enter the RUW (Read-Update-Write) state where it reads the current vertex value from the memory, updates the value with the delta of the current event through the FPU, and writes the updated value back to the memory. After finishing RUW, the processor will enter EVGEN where it reads adjacency information of the current vertex from the memory, compute the propagation delta of the current event based on the adjacency list, and generate propagation events based on the adjacent vertices and the propagation delta. Once all events are properly received by the event queue, the processor will return to the IDLE state and signal to the event queue that it is ready to receive a new event. Our event processors are efficient because as can be seen below in Fig. 8, the RUW and EVGEN operations do not have any inter-dependencies. Their operations are therefore highly parallelized because of this minimized data dependency.

Currently, we have four event processors on our chip. Since all event processors work in parallel and independently from each other, there is no theoretical limit to the number of event processors that we can have on our chip. However, when the number of processors on-chip scales up, the upstream and downstream crossbars and interconnects will become prohibitively large in the physical implementation. We decided to go with four event processors in order to keep the complexity of our interconnects manageable.



IDLE
VC READ: Current Vertex Value
FPU: damping factor * delta
EC READ: start and end
RUW
FPU: curr_vertex_value + delta
FPU: (damping factor * delta) / num_of_adjacent_vertex
EVGEN
VC WRITE: Updated Vertex Value
EC READ: Adjacency list
IDLE
To Top

Fig. 8: Event Processor FSM

F. Memory and Memory Controllers

The original paper on GraphPulse architecture includes an on-chip cache with a prefetching module that looks at the incoming events into the event processors and prefetch accordingly to further accelerate the processing speed. We eventually decided against implementing an on-chip cache and prefetcher in order to control the complexity of our project. Instead, we implemented an interface to off-chip memory and used two off-chip Verilog memory models for storing vertex and edge/adjacency information. These memory models have fully configurable sizes, store 64 bits per address, and have a built-in latency of 5 clock cycles for every read and write access. On the vertex memory, we store one 16-bit vertex value per address in order to maintain the atomicity of our operations. On the edge memory, we store our adjacency matrix in the compressed sparse row (CSR) format and store up to four 16-bit row indices or eight 8-bit column indices per address to maximize spacial locality and reduce the frequency of memory access.

Since all adjacency list information and vertex values are stored in two off-chip memory models (one for adjacency list and one for vertex values), we need two memory controllers (one for each memory) to decide which processor gets to access the memory in the current cycle and ensure that access to memory is properly shared between the processors operating in parallel.

The memory controller is simply a round-robin arbiter surrounded by some additional combinational logic to select the system's output to memory in the current cycle. The memory controller will see which event processor has valid read/write requests and grant access to one of the requesting processors. Once an event processor is granted access, it will capture the tag returned by the memory in the next cycle. When the memory returns valid data, the processor will use that tag to identify the fulfillment of the previous memory request.

G. Crossbar from Event Processors to Event Queues

Because of the irregularity of connections between vertices, this crossbar must be able to route any inputs (from the PEs) to any outputs (to the bins in Event Queues), while maintaining the fairness of events. As in the logical diagram for this crossbar, the input requests to the same output are first routed to the arbiter via switches, and then the round-robin arbiter grants one of the events to get through. As there are eight event generation streams in the PEs and eight bins in Event Queues, an 8x8 crossbar is implemented.

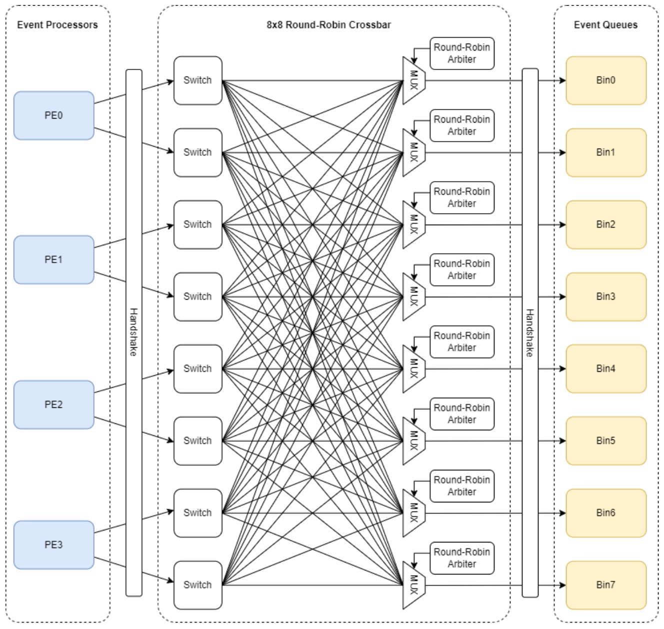


Fig. 9: 8x8 crossbar with round-robin arbitration from Event Processors to Event Queues

IV. Design Flow & Results

A. Golden Brick & Verification

We implemented our cycle-accurate golden brick model in Python. The file io_port.py has global variables signal, signal_n for the output of all the modules. In each module class, there's a one_clock() method, where we update io_port.signal_n through some logic operation. In the main function of the golden brick model, we have an update() function, where every signal is updated by the signal_n value. This is to emulate the flop flops in the circuit. With the golden brick in this style, we can transfer it into Verilog code easily. However, it caused inefficiency in development because we are basically doing the exact same thing in both the golden brick model and RTL code. Therefore, any changes to the circuit design require modifications to both models.

We used several random simulation vectors to check the functionality of the golden brick model. However, we used a single simulation vector for verification in RTL, post-synthesis, and post-apr stages due to the longer verification period in those stages. We chose an interesting vector as shown below, where the 10 vertices are pointing to each other in a circular way. The golden output for this input vector is all the vertices ending up with 0.1, and we obtained the same result in both RTL simulation and gate-level simulation (clock cycle: 22ns).

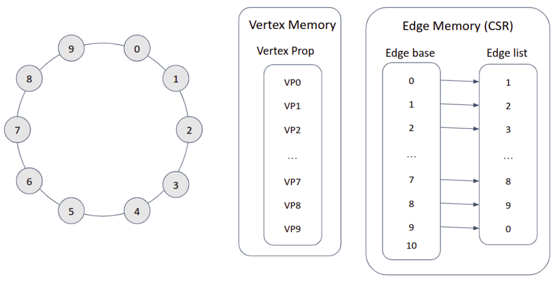


Fig. 10: A 10-vertex circular graph test vector, and vertex/edge memory contents

One interesting issue we encountered when doing the gate-level verification is that we initially had many x values in the netlist even though the synthesis reports say that there's no negative slack. We later realized that this was caused by the x propagation in the netlist since we assigned x to some registers, and this bug cannot be discovered in RTL because x values will not propagate in RTL code but only in the netlist. We ended up fixing the RTL code and re-synthesizing them to pass the post-synthesize simulation.

Another interesting issue that we encountered during post-synthesis verification is that the post-synthesis netlists enter undefined states in the simulation despite the error-free synthesis report and the error-free RTL simulation. Upon further inspection, we realized that in our RTL code, we overused combinational nets in the conditional statements of the "if ... else ..." statements in our large "always_comb" block for our finite state machines as a result of a direct mapping to Verilog from our goldenbrick code. Using combinational nets as conditions generated several long and degenerate combinational paths throughout our designs, which led to extremely unpredictable timing behaviors that were exposed during our post-synthesis simulations. This issue was resolved after changing these conditional statements to stable and clocked/registered values, and the overall timing behavior across our design was improved.

B. Synthesis and Place & Route result

To minimize the overall datapath, we divided the event queues into 2 groups, where 1 group contains 4 bins. As shown in Fig. 11, the 4 PEs and 2 EQs are of similar distance to the shared resources, and we grouped the shared resources together to get better area efficiency. The gaps are reserved for top-level routing.

However, we later realized that grouping all the shared resources would lead to DRC error when performing APR since the overall density is too low. Therefore, we divided the flattened center modules back into several sub-blocks, as shown in Fig.12. Also, for better routing to and from pads, modules are placed with more gaps in between. The final width

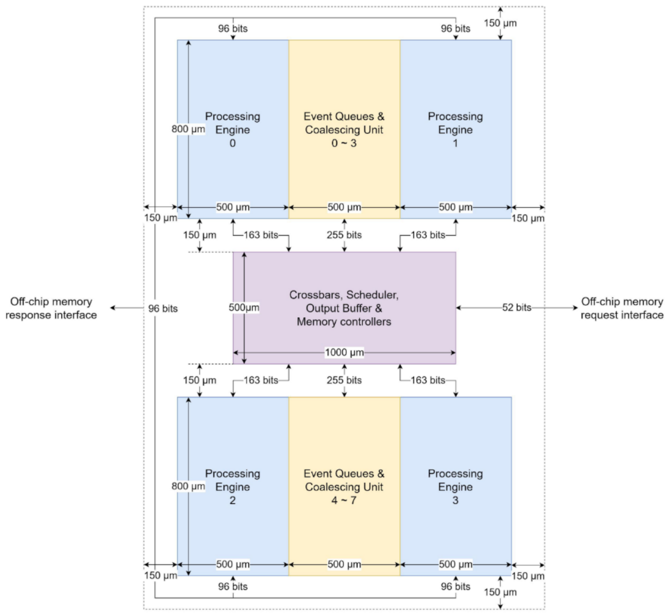


Fig. 11: Original floorplan: all the shared resources are grouped in the center module

and height (including the pad ring) are $3.3mm$ and $4mm$, respectively, and the total area is 13.2 mm^2.

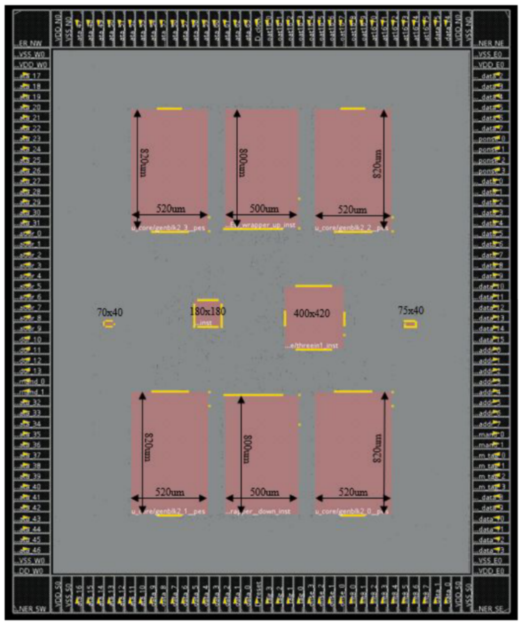


Fig. 12: Final floorplan: the shared resources are divided into 4 submodules

All the submodules' APR are DRC/LVS clean in Virtuoso. The top-level apr result figures are shown in Fig. 13, where the violations are all antenna violations in Innovus, meaning the pad ring, floorplan, and routing are feasible. However, our top-level integration has DRC/VLS errors in Virtuoso due to the power grid generation. All of the sub-modules are configured to use M1 M6 for routing, but the power grids are mistakenly placed on M4 and M5, so the top-level APR cannot correctly connect the top-level power grid (M7 and M8) to sub-module power.

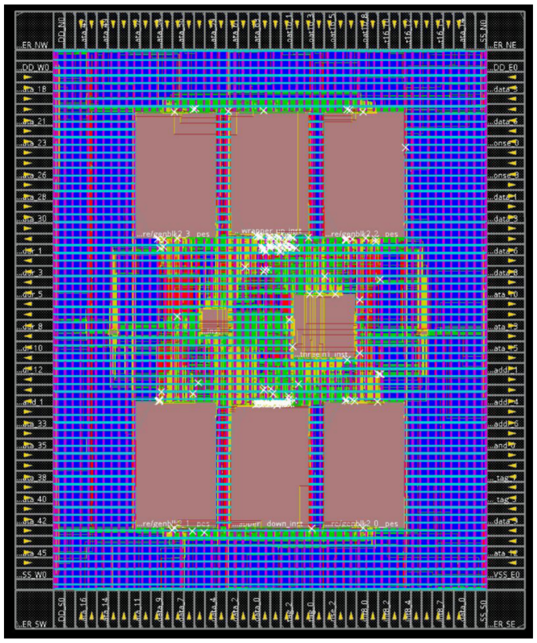


Fig. 13: APR result of the top-level

The power results are shown in Table I, and the place & route metrics are shown in Table II.

Module	Power			
	Switch/mW	Int/mW	Leak/pW	Total/mW
Event Queues (4 bins)	1.055	5.044	5.363+06	6.105
Scheduler + Output Buffer + 4x4 crossbar	0.571	1.362	1.13e+06	1.934
Event Processor	6.426	5.726	1.31e+07	12.165
Edge Mem Ctrl	4.24e-02	2.87e-02	1.82e+04	7.11e-02
Vertex Mem Ctrl	6.06e-02	3.22e-02	2.40e-04	9.28e-02
8x8 crossbar	0.29	0.762	5.89e+05	1.052
Top	0.728	19.549	4.75e+07	20.325

TABLE I: Post-APR power for sub-modules and post-synthesis power for top-level

Module	Area/μm^2	Density	Clock /ns	Setup /ns	Hold /ns
Event Queues (4 bins)	500×800	52.48%	10	0.289	0.066
Scheduler + Output Buffer + 4x4 crossbar	400×420	40.10%	10	0.431	0.105
PE	520×820	61.16%	10	0.431	0.105
Edge Mem Ctrl	70×40	75.84%	10	8.338	0.179
Vertex Mem Ctrl	75×40	96.93%	10	8.216	0.178
8x8 crossbar	180×180	95.97%	10	5.001	0.061
Top	3300×4000	-	11	4.096	0.211

TABLE II: Place & route metrics for submodules and top-level

V. Conclusion

In this project, we focus on the design and implementation of the GraphPulse accelerator to run PageRank. A Python-based cycle-accurate golden brick is first constructed from scratch to verify the microarchitecture of the whole system. Although it takes more time and effort compared to coarser-grained simulators, this methodology is proved to be very efficient to generate randomized unit tests and debug the RTL design. The VLSI implementation is done in a hierarchical manner. All the post-APR sub-modules in the system are DRC/LVS clean. The top-level passes the post-synthesis simulation at the clock frequency of 22ns, with all the sub-modules in the post-APR state. The post-synthesis power estimation of the whole chip is 20.325mW. Due to a mistake in the sub-module power grid design and limited time, the top-level APR is not successful, but the proposed top-level floorplan, pad ring, and routing are proved to be feasible. The dimension of the whole chip is $3300\mu m \times 4000\mu m$.

VI. Group Dynamics

The workload distribution among the group is shown in TABLE III.

Member	Work
Ganrun Xu	Event Queues, In-place Coalescing Unit, Scheduler
Wenhan Kou	Event Queues, In-place Coalescing Unit, Scheduler
Peter Zhong	Event Processors, Floating Point Unit, Memory Controller
Xiuji Wu	APR of all submodules and top-level APR
Yu-Sheng Ting	Golden Brick framework, top-level integration and APR
Zetong Guan	Crossbars, Event Queues, top-level APR and debug

TABLE III: Group Dynamics

References

[1] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 908–921.