Magic Memory: A Programming Model For Big Data Analytics

Eric Tang, Franz Franchetti
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, United States
{erictang, franzf}@andrew.cmu.edu

***Abstract*—Big data analysis is a difficult task because the analysis is often memory intensive. Current solutions involve iteratively streaming data chunks into the compute engine and re-computing the analytical algorithm. To simplify this process, this work proposes a new programming model called Magic Memory. In this model, persistent invariants or functional dependencies can be established between memory regions. These functional dependencies will always be held true when this memory region is being read. Recent technological advancements enable Magic Memory at the hardware level, providing performance that is hard to achieve with a software-only solution. Our ongoing work seeks to explore an implementation of Magic Memory on a CPU-FPGA system, where the CPU runs the host code while the FPGA provides hardware acceleration. The CPU allocates memory on the FPGA and declares an invariant for the FPGA to uphold on this region of memory. We demonstrate how an application such as PageRank can utilize Magic Memory to recalculate its output as the input graph is modified automatically.**

***Index Terms*—FPGA, Programming Model, Shared Memory**

I. Introduction

Interactive data analysis is a difficult task given the amount of expert insight and computing power that is necessary. This process often begins with a domain expert exploring a wide range of questions with the goal of gleaning specific insights. An expert must formulate a question about the dataset, modify the dataset, and perform the computation necessary to get the desired result. This process is often repeated many times which is slow and distracts from the expert's main focus of understanding the dataset [1]. Therefore, this paper introduces Magic Memory, a programming model with an intuitive API for data analysts which can leverage hardware accelerators to reduce the computation time of complex data analytics algorithms.

II. Proposed Solution

Magic Memory is the idea that one can declare a persistent relationship between regions of memory. When a write occurs to the input memory region, the values in the output memory region will be updated accordingly such that the relationship is maintained. This mindset is similar to how one declares formulas between cells in Excel. When any one input cell's value is changed, the dependent output cells will automatically be updated with the new value. For example, in Magic Memory, if the input memory region is a matrix, the output memory region can be declared to hold the sum of each row of the matrix. Whenever a write occurs to an element in the input matrix, the sum of the row where that element resides will be recalculated and the output vector will be updated. If a read to a Magic Memory region occurs before the output is computed, the response will be blocked until the computation is complete. Figure 1 illustrates the data flow of a user interacting with Magic Memory and Figure 2 shows a code snippet for this program. Since Magic Memory can be accessed with read or write instructions, users can utilize this model along with state-of-the-art hardware acceleration with little change to the original program.

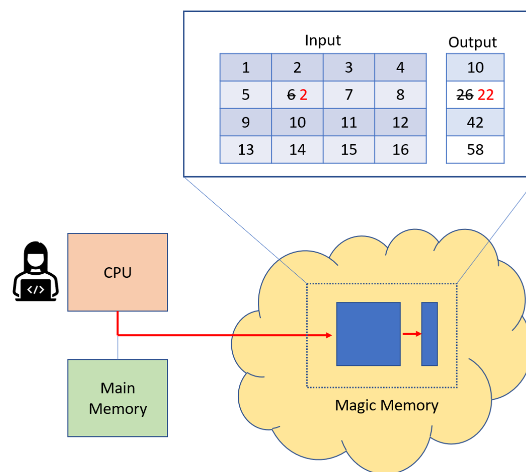

Fig. 1. A user set up the output memory region to be a vector that calculates the sum of each row in the input matrix. When the user updates the input region with a new value, the output is automatically updated. An important detail that can be seen here is that the Magic Memory regions may not be stored in main memory.

This event-based programming model can be utilized on CPU-FPGA systems as a simple way to interface with hardware accelerators. On the CPU, a program can access Magic Memory by reading and writing to a specific set of addresses that are mapped to the FPGA. This invokes the hardware accelerator. The hardware accelerator then performs the appropriate calculations and reads or writes the necessary addresses in order to maintain the persistent relationship desired by the program.

Implementation. As a proof of concept, this programming model has been implemented on an Intel i7-10700 CPU

```
int magic_memory_add(){

int N = 5;

// A 0 terminated array
int input_dims[3] = {N, N, 0};
int output_dims[2] = {N, 0};

uint64_t* input = magic_mem_in_alloc(input_dims);

// Output configured to calculate the sum of
// the row
uint64_t* output =
        magic_mem_out_alloc(output_dims);

// Update performed with simple array accesses
input[3] = 5;
input[4] = 6;

uint64_t row0_sum = output[0];

return row0_sum;
}
```

Fig. 2. A simple Magic Memory program to sum the rows of a matrix

running Linux with a Stratix 10 MX FPGA connected via PCIe Gen 3. The inputs and outputs to Magic Memory are declared by calling a Magic Memory malloc function. This function returns an illegal address by taking advantage of the fact that x86 virtual address space only uses 48 of the 64 available bits. The extra 16 bits are then used to identify each Magic Memory region that is allocated, with each region having 256TB of addressable memory. This enables Magic Memory to support a dense address space for very large graphs with millions of nodes. Any attempt to read or write these regions will trigger a segmentation fault and will invoke a custom segmentation fault signal handler. This handler reads the 16 most significant bits in order to identify which Magic Memory region is being read. The handler goes on to decode the violating instruction and extracts relevant information including - the memory transaction type (read or write), the address being accessed, and the register to store the result to (if read instruction). This information is then written to a memory-mapped FIFO on the FPGA.

The FPGA will also maintain storage of the input and output Magic Memory regions. For instance, while the user may access very large sparse matrices using dense notation, the FPGA can store the matrix using any sparse representation. This allows for programs to avoid the large amount of complexity necessary for maintaining sparse data structures and the need to modify the program to support various sparse formats. In addition to handling how data is stored, this implementation of Magic Memory uses a hardware accelerator to perform the necessary computation to maintain the invariant. The FPGA interprets requests to these memory regions by first identifying the type of request and where to read or write the data. On a read request, the FPGA responds with the appropriate data from the corresponding location by writing to another memory-mapped FIFO. If the CPU sends a read request to an output region that is still being computed, then that request will be blocked until the value is ready. On a write request, the data is written to the appropriate location and then a check is performed to see if the output must be recalculated. If so, the compute kernel will be invoked, and the output will be updated when the computation is complete. Finally, attempted writes to the output region are ignored since the output region is solely dependent upon the values in the input region.

III. Results

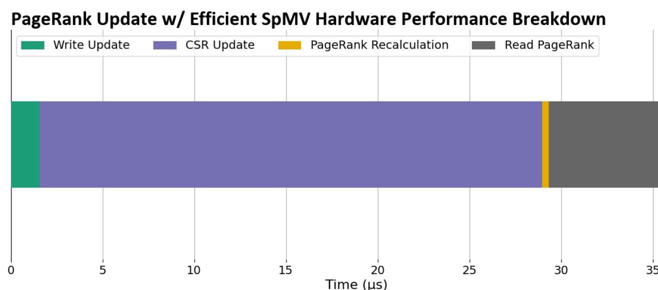

Fig. 3. Using an efficient SpMV hardware on the FPGA, a breakdown for the time to recalculate the PageRank of a graph after adding/removing a single edge can be seen above.

Early implementations of PageRank [2] on FPGA allow for using Magic Memory on some small graphs. By modeling the results from more efficient sparse matrix-vector hardware on FPGA like the one described in [3], the time to compute an update to Magic Memory can be seen in Figure 3. From this, it is clear that with future improvements to the shared memory model between the CPU and FPGA, updates to these memory regions could happen in the time that it would take to service a TLB miss.

IV. Future Work

Magic Memory will provide users with a simple programming model and an easier method for performing interactive data analytics. The next step to realizing this will be to shorten the design time required to program the FPGA.

Current limitations, such as the need to design custom hardware accelerators, inhibit the fast adoption of this programming model. We believe that future improvements to the shared memory model as well as support for generalized overlays will enable widespread adoption of Magic Memory with performance comparable to state-of-the-art.

References

[1] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. Haas, "Interactive data analysis: the control project," *Computer*, vol. 32, no. 8, pp. 51–59, 1999.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[3] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–358. [Online]. Available: https://doi.org/10.1145/3352460.3358330