Magic Memory: A Memory-Centric Declarative Programming Paradigm to Enable High Productivity on Heterogeneous Systems

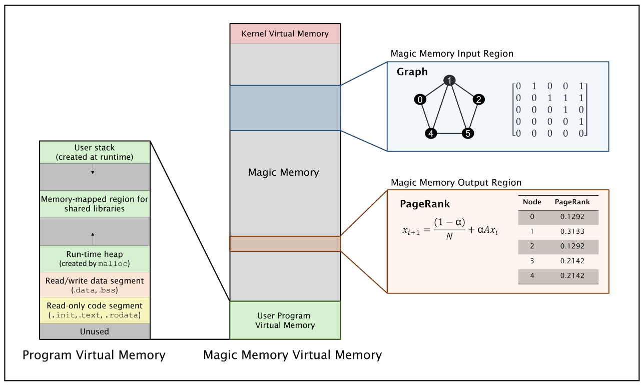
Eric Tang, James Hoe, Franz Franchetti
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, United States
{erictang, jhoe, franzf}@andrew.cmu.edu

***Abstract*—Heterogeneous computing systems involving CPUs, GPUs, FPGAs and even ASICs have already been shown to have immense computational power for various applications. However, programming these systems efficiently and effectively remains a significant challenge. Traditional imperative programming approaches result in complex programs that require expert-level understanding in order to optimize or maintain. The declarative programming model, Magic Memory, focuses on the behavior of the program and thus hides the nature of the underlying hardware that executes the program. In this model, computation is expressed using mathematical functions between arrays. This function can then be treated as an invariant that is always held true. During steady state, computation is performed by observing changes to these magic memory regions and then only recomputing the values that are affected by the change. As a first demonstration of this work, we utilize a CPU-FPGA system to compute the PageRank of a graph and recompute the PageRank after adding or removing an edge from the graph.**

I. Introduction

Heterogeneous computing systems, which can be composed of hardware such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs), are renowned for their exceptional computational capabilities across a wide range of applications. These systems leverage the unique strengths of each component to achieve superior performance and efficiency. Despite their potential, the effective and efficient programming of these systems can be challenging. Traditional imperative programming approaches often result in complex and convoluted code, demanding a high level of expertise for optimization and maintenance.

The complexity of programming heterogeneous systems has spurred interest in alternative programming models that can simplify the development process while harnessing the full power of the hardware. One promising approach is the declarative programming model, which focuses on the behavior of the program rather than the intricacies of the underlying hardware. This paper introduces ”Magic Memory”, a declarative programming model designed to abstract the details of hardware execution. By expressing computation through mathematical functions between arrays, Magic Memory allows developers to define program behavior as invariants that are consistently maintained.


Kernel Virtual Memory
Magic Memory Input Region
Graph
0
1
2
4
5
$\begin{bmatrix}0&1&0&0&1\\0&0&1&1&1\\0&0&0&1&0\\0&0&0&0&1\\0&0&0&0&0\end{bmatrix}$
User stack (created at runtime)
Memory-mapped region for shared libraries
Run-time heap (created by malloc)
Read/write data segment (.data, .bss)
Read-only code segment (.init, .text, .rodata)
Unused
Magic Memory
Magic Memory Output Region
PageRank
$x_{i+1} = \frac{(1-\alpha)}{N} + \alpha A x_i$
User Program Virtual Memory
Program Virtual Memory
Magic Memory Virtual Memory

Node	PageRank
0	0.1292
1	0.3133
2	0.1292
3	0.2142
4	0.2142

Fig. 1: For x86 systems, Magic Memory regions utilize the virtual memory addresses outside of the user space and the kernel space. Here, the input Magic Memory region is a dense matrix which represents a graph while the output Magic Memory region is a vector with the PageRank of each node of the graph.

Magic Memory enables programmers to describe computation using mathematical functions between regions of memory. Input Magic Memory regions store data used as input to the Magic Memory function, while output Magic Memory regions store the function's output based on the input region's values. When new data is written to the input memory region(s), the values in the output memory region(s) are updated to maintain the invariant. By describing the computation between memory regions, the input and output data can be stored in various places across the system, and this complexity can be hidden from the user. The mathematical basis of the function describing the relation between the memory regions allows a code generation system, like SPIRAL [1], to understand the computation and generate a program or hardware for a wide range or architectures.

Figure 1 illustrates the virtual memory allocation of a Magic Memory program in an x86 system, and Listing 1 shows a code snippet for this program. The user virtual memory space only uses the lower 48 bits of the 64 available bits in the x86 system with the kernel virtual memory only using a small portion of those upper bits. Magic Memory takes advantage of this and is allocated in the unused address space of x86 systems. This large address space that is available enables dense addressing for large sparse data structures. Then, as described later on, these addresses are able to be translated and utilize a wide range of sparse data structures. Furthermore, since Magic Memory regions are allocated on the traditional virtual memory stack, these regions can be relocated across devices, allowing users to utilize various hardware accelerators with minimal change to the original program.

As a first example, this programming model is realized for the popular PageRank [2] algorithm using an FPGA. In the PageRank algorithm, the graph nodes are assigned a rank based on the number of edges coming in and out of each node. The graph is allocated using a Magic Memory input region, the ranking for each node is allocated to a Magic Memory output region, and the relation between the two regions is defined as the PageRank algorithm. This enables data analysts to observe the effects of adding or removing an edge from the graph from a single store to memory. In the magic memory input region, the sparse nature of the graph is hidden from the programmer, and a single load or store to memory is needed to update the graph data structure. The magic memory output region is allocated on the FPGA allowing for a specialized sparse matrix vector multiplication (SpMV) accelerator to be leveraged to perform the primary computation in the PageRank algorithm. In this example, the algorithm implemented using Magic Memory benefits because of its iterative nature, the stream of updates to a sparse data structure, and the frequency of these updates.

```c
int** _mx_graph = (int**)_mx_malloc(N*N*sizeof(float));
float* _mx_pagerank = (float*)_mx_malloc(N*sizeof(float));

_mx_memcpy(vec_mx, graph, NNZ*sizeof(float));

// Set up relationship between Magic Memory regions
_mx_setup(_mx_pagerank, algorithms::page_rank, _mx_graph);

// Check initial PageRank of node 3
printf("Node 3 PageRank = %f\n", _mx_pagerank[3]);

// Add an edge between nodes 3 & 4
_mx_graph[3][4] = 1;

// Check the new PageRank
printf("New Node 3 PageRank = %f\n", _mx_pagerank[3]);
```

Listing 1: PageRank Algorithm using Magic Memory. Here the GraphBLAS Template Library (GBTL) [3] PageRank algorithm is used to describe the invariant between Magic Memory regions

II. Methodology

Software. Magic Memory is made accessible to the programmer through a simple header-only C library designed to mimic the C standard library. The input and output Magic Memory regions are allocated using Magic Memory equivalent versions of the malloc and calloc functions. This function returns an illegal address by taking advantage of the fact that x86_64 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 sparse data structures. These regions can be initialized through a Magic Memory memcpy that not only copies the data from the source address but also initializes the relevant Magic Memory metadata. Any attempt to read or write a Magic Memory address triggers a segmentation fault and invokes 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 type of memory transaction (read or write), the address being accessed, and the register to store the result (for read instructions).

Key metadata on each Magic Memory region is stored in a struct that is created when each region is allocated. This metadata contains information regarding the size, whether the region is sparse, where this data is actually stored, which other Magic Memory regions depend on the values from this region, and what the relationship is between the Magic Memory regions. Upon each Magic Memory load or store that occurs, a segmentation fault occurs due to the illegal address that is being accessed. When the Magic Memory signal handler decodes the index and region that the original program attempted to write, the system references the corresponding struct to glean enough information to update the appropriate sparse or dense data structure and can propagate this information across the system as needed.

Hardware. As a proof of concept, Magic Memory is implemented on an Intel i7-10700 CPU running CentOS 6 with a Stratix 10 MX FPGA connected via PCIe Gen 3. In addition to performing the computation of the invariant between Magic Memory regions, the FPGA can own the 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 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. If it is a read request, the FPGA simply responds with the appropriate data from the corresponding location by writing to another memory-mapped FIFO. If the CPU sends a read request to the output region before it is ready then that request will be blocked until the computation is complete. If it is a write request, the data is written to the appropriate location and then the input is checked 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 due to the fact that the output region is solely dependent upon the values in the input region.

III. Example Application: Interactive PageRank

PageRank is a common algorithm used by search engines to rank web pages in search results. This algorithm uses the links coming to and from each page and the quality of the pages that the links come from in order to create the ranking. In PageRank, the input region is the graph of hyperlinks in matrix format with each element being the probability of jumping from node to node. An important property of a hyperlink matrix is that the sum of every column must be one. Therefore if a node (column) has no outgoing edges (elements within a column), that column will have a uniform distribution such that this property is maintained.

The PageRank algorithm then iteratively performs a **Sp**arse **M**atrix dense **V**ector (SpMV) multiplication. The matrix representing the graph is multiplied by the current PageRank and the PageRank is updated with the output of this operation. This process is repeated until the difference between each iteration is less than some small epsilon. The output memory region stores the rank of each node in the input graph. While SpMV is a computation common in many graph applications, efficient SpMV is difficult due to the large amount of accesses to random memory locations. For this reason, a high performance hardware accelerator on the FPGA would greatly improve the performance of this application along with many more.

Magic Memory Implementation. With Magic Memory, the graph is allocated as a matrix in a Magic Memory input region and the PageRank vector is allocated as a Magic Memory output on an FPGA. When writing the Magic Memory input region, the memory address is broken down to determine the row and column index of the data access. The row and column index are then used to index into the sparse data structure that is used to store the graph. By hiding the sparse data format from the programmer, this introduces a level of separation between the application description and the software or hardware implementation that is used behind the scenes in order to achieve good performance. The core computation of PageRank (SpMV) is implemented in hardware on the FPGA to take advantage of the sparse data format. The Compressed Sparse Row (CSR) representation of the matrix, the read pattern of the data and column arrays follows a simple sequential pattern which lends itself to using a burst coalesced Load-Store Unit (LSU). This LSU buffers contiguous memory requests for the largest possible burst. In order to utilize this hardware, each row is stored as a chunk of elements of a fixed size with the excess entries simply set to 0. By using a set size, the loop that iterates over the nonzeros of each row is able to be fully unrolled at compile time with no extra control logic inside the loop needed to check to see if the end of the row has been reached. This also allows for the LSU selected to have a bandwidth that matches the number of entries needed to fill the fully unrolled loop. Since each element of the output vector can be computed independently, the loop that iterates over these elements is pipelined and a loop iteration is able to be scheduled every cycle.

	Read (μs)	Write (μs)
Memory-Mapped Access	5.46	0.54
Magic Memory	6.6	1.57
Difference	1.14	1.03

TABLE I: Memory Mapped FIFO Latency

IV. Evaluation

We measure the overhead of the Magic Memory abstraction by performing a memory-mapped FIFO latency test. This involves measuring the read and write latency to the memory-mapped FIFO on the FPGA for one million accesses. Table 1 shows the latency from directly writing to the memory-mapped FIFOs versus the latency from writing to the FIFOs with the Magic Memory programming model. These results show that an additional 3000 cycles are required for a read or write to an address with the Magic Memory abstraction.

While this is not ideal, it is comparable to other common memory accesses such as a TLB miss or a lower level cache miss. Additionally, this overhead can be minimized by utilizing interrupts instead of signals. When invoking the signal handler, the CPU must switch from user mode to kernel mode and then back again before the program can resume. By utilizing an interrupt, this expensive switching between kernel mode and user mode could be avoided and this overhead can be reduced dramatically. Finally, one can directly boot the machine in kernel mode. While this is not ideal from a security standpoint, it does solve the problem of the need to switch between different operating modes.

Comparison to CPU Baseline. The FPGA being targeted for this application is a Stratix 10 PAC card with 32 GB of DDR4 memory. As a comparison to Magic Memory, we use a CPU implementation that also computes the SpMV using the CSR representation of the matrix. This program is run on an Intel Xeon Platinum 8256 as a baseline comparison.

Various different types of datasets were used to evaluate the FPGA design. The first example utilizes a graph with 100k nodes, where each node has 32 edges. This graph is constructed in such a way that it perfectly fits the hardware generated for the FPGA and serves to provide an upper bound on the performance that can be expected from hardware. Following this, other graphs with other average degrees are used to investigate how this kernel fairs for graphs that do not always use the maximum bandwidth to load useful data. In Figure 2, the runtime of SpMV for various different graph sizes is shown. Due to the nature of the hardware, the number of elements in each row of the matrix does not affect the running time of the FPGA. For this reason, for a graph where the maximum number of edges for each node is less than the block size (32 in this case) and a set number of nodes, the runtime is constant. This can be observed in the figure where the blue bars are a constant height. For the CPU implementation, the average degree of the graph is directly

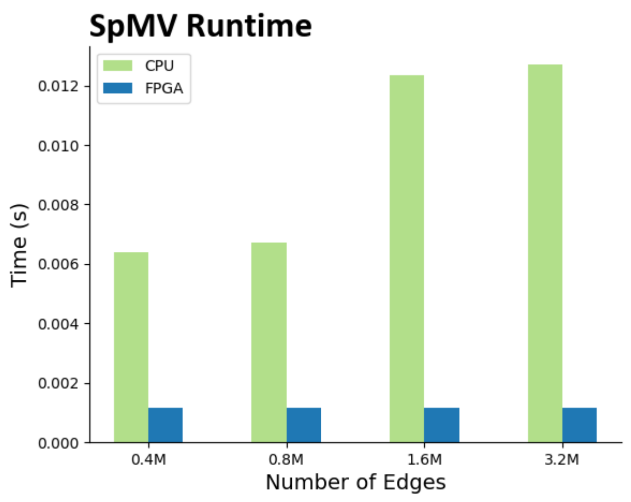


Fig. 2: Comparison of SpMV kernel on FPGA versus baseline CPU implementation. All graphs used to generate the sparse matrix have 100,000 nodes which makes the SpMV operation a 100k x 100k sparse matrix multiplied by a 100k dense vector. By varying the number edges in graph, the sparsity of the matrix is changed

correlated with the runtime of the graph. However, even with graphs of an average degree of 4 the FPGA implementation is still able to provide a near 6x improvement over the baseline CPU implementation.

V. Related and Future Work

Magic Memory provides users with a simple programming model for programming heterogeneous systems. In order to realize this, the design time for writing and designing hardware for the FPGA must be shortened. In the current implementation, designing and testing a hardware accelerator required a large amount of time. However, research efforts on creating domain-specific FPGA overlays could address this issue. The Gorilla (now called Primate) project creates an overlay generator to create a soft processor with specialized functional units for various applications [4]. Using this tool, an efficient hardware accelerator can be created on the FPGA for an invariant declared by a C++ program.

Other algorithms that could benefit from Magic Memory in the future will require a different API for accessing data. While a dense addressing mode is beneficial when trying to perform random reads and writes into the graph, many other graph algorithms follow a more regular access pattern that is based on each node or edge or even the neighbors of each node. Furthermore, the structure of the sparsity greatly affects the optimal algorithm and corresponding data formats for a given graph algorithm. In breadth first search (BFS), the average degree of the node determines whether a top-down or bottom-up algorithm yields better performance [5]. Using Magic Memory to use a declarative programming model to describe BFS without constraining how this computation is performed, the appropriate data format and computational algorithm can be used with minimal changes to the original source code.

The current bottleneck of streaming Magic Memory requests to the FPGA will be addressed as future technologies emerge. CXL is an upcoming technology that allows for fast cache-coherent memory across PCIe Gen 5. This allows for Magic Memory regions to be allocated in the FPGA NUMA domain and utilize shared memory to communicate with the CPU. The FPGA can then be designed to intercept Magic Memory requests and eliminate the need for a signal handler.

VI. Conclusion

The Magic Memory programming model can be used to effectively describe computation for a heterogeneous system. By using a declarative programming paradigm to describe computation at a high level between virtual memory regions, a separation between the algorithm description and the hardware computation is introduced. The application programmer is able to focus on the algorithm in question and does not need to be an expert hardware engineer in order to make use of the resources available, Meanwhile, the system is able to allocate the data to the appropriate hardware target and can utilize programs for whichever architecture best fits. In this work, it is shown that Magic Memory can be useful for specific data analysis algorithms such as PageRank. However, current limitations, such as the need to design custom hardware accelerators, inhibit the fast adoption of this programming model. With future improvements to the shared memory model across heterogeneous systems with CXL, as well as support for generalized overlays making FPGAs increasingly more programmable, the Magic Memory programming paradigm will be able to more effectively leverage the capabilities of state-of-the-art accelerators.

VII. Acknowledgments

This work is supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, “Spiral: Extreme performance portability,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[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] “Graphblas template library (gbtl), version 3.0,” Available at https://github.com/cmu-sei/gbtl, June 2020.

[4] M. Lavasani, L. Dennison, and D. Chiou, “Compiling high throughput network processors,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 87–96. [Online]. Available: https://doi.org/10.1145/2145694.2145709

[5] S. Beamer, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Washington, DC, USA: IEEE Computer Society Press, 2012.