A Synthesis Methodology for Application-Specific Logic-in-Memory Designs

H. Ekin Sumbul, Kaushik Vaidyanathan, Qiuling Zhu, Franz Franchetti, Larry Pileggi
Electrical & Computer Engineering Department, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213 USA
{hsumbul, kvaidya1, qiulingz, franzf, pileggi}@andrew.cmu.edu

ABSTRACT
For deeply scaled digital integrated systems, the power required for transporting data between memory and logic can exceed the power needed for computation, thereby limiting the efficacy of synthesizing logic and compiling memory independently. Logic-in-Memory (LiM) architectures address this challenge by embedding logic within the memory block to perform basic operations on data locally for specific functions. While custom smart memories have been successfully constructed for various applications, a fully automated LiM synthesis flow enables architectural exploration that has heretofore not been possible. In this paper we present a tool and design methodology for LiM physical synthesis that performs co-design of algorithms and architectures to explore system level trade-offs. The resulting layouts and timing models can be incorporated within any physical synthesis tool. Silicon results shown in this paper demonstrate a 250x performance improvement and 310x energy savings for a data-intensive application example.

Categories and Subject Descriptors
B.7.1 **[IC]:** Types and Design Styles - *memory technologies*

General Terms
Design, Performance

Keywords
Application specific synthesis, embedded logic-in-memory, smart memory, SRAM, SpGEMM.

1. INTRODUCTION
Embedded memory occupies more than 50% of the overall chip area in a modern SoC (System on Chip) design [5]. Large compiled memory blocks with mostly-square aspect ratios are the most area efficient choice to minimize the periphery and lithographic area penalties, however, for data-intensive applications, most of the performance and energy is spent on transferring on-chip data over long distances. One solution to address this problem is to co-optimize the algorithm, architecture, and hardware, wherein finely tailored small memory arrays, integrated with application-level knowledge, are used to localize the computation and minimize the data transport over long distances. Such application-specific targeting, however, generally incurs a high cost of custom design. Alternatively, compiling small granular arrays of embedded memory blocks following a conventional ASIC approach results in area inefficiency. This suggests the need for a design methodology that effectively addresses the challenges of memory-intensive applications while maintaining an affordable design cost.

One enabler for creating such a methodology comes from scaling trends of sub-20nm technology nodes. Restrictive patterning in deeply scaled nodes constrains designers to map memory and logic to a small set of well-characterized layout patterns [8]. Therefore, while memory compilers have been traditionally used to assemble hard IP layout slices of bitcells and periphery, restrictive patterning makes the patterning the only critical "hard IP." Moreover, while restrictive patterning is seemingly an impediment for layout efficiency, it does provide an opportunity to place memory cells and random logic gates in close proximity without creating lithographic hotspots, thereby avoiding area spacing penalties. Leveraging this technology constraint as a "feature," the opportunity for synthesis of smart memory blocks was proposed in [6], wherein specialized computation logic and embedded memory could be tightly integrated for localized computation and energy savings. With lithographically compatible memory and logic cells, any application specific customization can be reliably synthesized into the embedded memory block.

The energy and performance benefits of localizing computation for data-intensive applications are well known [2][3][7][11][13]. There are various examples of "processor-in-memory" designs wherein processing units are placed in memory abstraction [2][3], or more recently, near data computing in 3D and 2.5D stacks to provide more bandwidth with less energy [11]. It has also been shown that various data intensive applications highly benefit from LiM blocks [12][13]. However, there remains the need for physical synthesis (rather than compilation) of LiM blocks that are compatible with a full chip physical synthesis and offer the ability of system-level exploration.

We propose a LiM methodology whereby the memory arrays are physically synthesized from "*memory bricks*" that represent the lowest level of physical abstraction (analogous to standard cells). The logic cells and memory peripherals are comprised of standard logic gates that are lithography-pattern compatible with the memory bricks. With memory bricks and logic cells at the same physical abstraction level, the conventional black box memory block becomes a "white box" since the boundary between memory and logic disappears. This enables memory arrays to be distributed in a fine-grained manner, thereby reducing signal travel distances and allowing the inside and outside of any memory block to be optimized across its boundary for performance, energy, and area. This automated synthesis further enables rapid design-space exploration for the overall system by generating pareto-curves of possible block designs.

We demonstrate our LiM synthesis methodology for the data-intensive application of Generalized Sparse Matrix - Sparse Matrix Multiplication (SpGEMM). SpGEMM is a core primitive in graph processing applications such as graph contraction or

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06…$15.00
http://dx.doi.org/10.1145/2744769.2744786

shortest-path algorithms, but traditionally cause very high unpredictable data traffic as the sparse matrices are structurally large and unpredictable [4]. We implemented two separate designs as low-power accelerators for SpGEMM algorithm in 65nm CMOS technology. A baseline design is implemented with conventional ASIC flow, and a LiM based design is implemented by using our proposed methodology where design-rule compliant memory bricks are generated by our automation flow. Both implementations use the same bitcells and have comparable area for a fair comparison. Silicon results show that our LiM based SpGEMM design provides dramatic benefits up to 250x speed-up and 310x better energy efficiency compared to the baseline design, coming from the co-design of algorithm and hardware.

In this paper, background on LiM design is provided in Section II, and our LiM synthesis flow is described in Section III. Section IV highlights details on the implemented SpGEMM system, and Section V provides analysis of the LiM based chip results. A final discussion on the overall flow is given in Section VI.

2. BACKGROUND

2.1 Restrictive patterning enablement

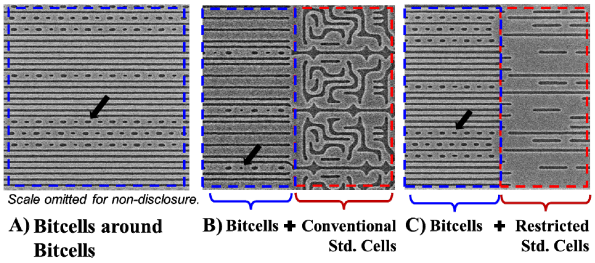

Figure 1. Metal-1 SEMS from a 14nm IBM test-chip showing bitcell printability when random logic is put next to bitcell arrays [10].

As CMOS scales to sub-22nm technology nodes with 193nm immersion lithography, design rules become increasingly complex and grow in number [8]. A pattern construct based layout design approach can provide an efficient and affordable interface between design and manufacturing by limiting the number of patterns used to construct a block design [6]. Then logic and memory bitcells can be constructed from a common set of pre-characterized layout patterns that are lithography compatible.

To demonstrate this patterning influence, Fig. 1 shows three Scanning Electron Microscope (SEM) images for Metal-1 layers corresponding to 14nm SOI FinFET based bitcells and logic standard cells as described in [10]. First, bitcell printability in the neighborhood of other bitcells is shown in Fig.1.a. Next, design rule compliant standard cells following conventional layout style are shown to hurt the printability in Fig.1.b. However, the regular patterned standard cells next to bitcells do not impact printability, as shown in Fig.1.c. These results validate that restrictive patterning can enable logic and embedded memory cells that are tightly integrated without requiring extra spacing for lithography compatibility.

2.2 Application-specific smart-memories

Customized smart-memories have been proposed and implemented for a varying set of applications to achieve the localized computation that leverages in-memory bandwidth for performance and saves energy by minimizing wasted data traffic. Data-intensive architecture (DIVA) [2] is a processing-in-memory system where multiple smart-memory co-processors are put in a traditional processor to accelerate several instructions by locally executing them within the smart-memory. Computational RAM (CRAM) [3] is another in-memory processing architecture wherein simple computational logic is pitch-matched and placed under memory columns, such that memory works either as a traditional memory or configured as a single instruction, multiple data (SIMD) smart-memory. Using this technique, parallel work is localized and in-memory bandwidth is used efficiently for various applications such as computer graphics, image processing, etc. Recently with the proliferation of 3D and 2.5D IC stacks and effective utilization of through silicon vias (TSV), near memory computation examples also leverage the high bandwidth that TSVs provide in between the stacked memories and the core as another degree of localization. For instance, "smart-3D" [11] architecture demonstrates how to maximize the useful data traffic in between stacked DRAM caches and the processor to minimize overall execution time. Alternatively, focusing on low-level aspects of the embedded memory, an SRAM can be implemented by choosing and tailoring its bitcells carefully for the target application as well.

There are also various examples that embed functionality into the SRAM to perform specific tasks for the application. A parallel access memory, for instance, is demonstrated in [7] for power efficient motion estimation. The parallel access memory stores a 2D image pixel array with a size of $K \times L$, and allows random access of pixels in a window of $m \times n$ in a single cycle (where $m<K$ and $n<L$). For a traditional ASIC approach, parallel access memory is realized by implementing logic blocks next to parallel accessible SRAM banks. 2D pixels are distributed to $m.n$ parallel memory banks for a conflict-free access, however, this does not exploit the address pattern commonality between the accessed pixels. Moreover, area and energy penalties are incurred when the image or access window size is large. For the same functionality, the smart-memory in [7] exploited the address pattern commonality and implemented an application-specific SRAM with shared and customized decoders. Row decoders are shared between m banks and customized to activate n adjacent wordlines with respect to single address. A column decoder is added under each bank group to select a single element per column from the activated multiple rows. With this customization, the same parallel access functionality can be handled inside the memory block with significantly less power and area.

Based on the same parallel access memory proposed in [7], a smart interpolation memory is proposed in [13] to accelerate the bottleneck of polar to rectangular grid conversion in Synthetic Aperture Radar application in an energy efficient way. The proposed interpolation memory is a LiM based seed table that uses a parallel access memory as a smaller seed table and interpolates the required data on the fly as if it is readily stored. Experimental simulation results in [13] show that an application specific LiM based architectures have great potential for improving the performance for such data-intensive applications.

3. LiM SYNTHESIS METHODOLOGY

While the design in [7] was an expensive manual customization, by leveraging the restrictive patterning for deeply scaled technologies we propose to fully synthesize logic and memory using the pattern constructs as the only necessary hard IP. By directly synthesizing application-specific functionality into the LiM block design in a fine-grained manner, smart-memories would not be custom designed at great cost, or compiled from hard IP slices at the expense of significant loss of performance and area. Moreover, with a fully automated synthesis approach, rapid design-space exploration and co-optimization of hardware and algorithm would be available for system level exploration.

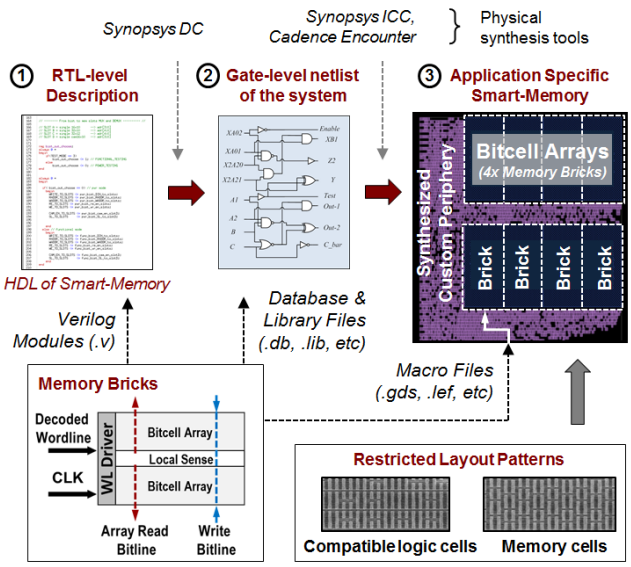

Figure 2. LiM synthesis flow: LiM system is synthesized from its RTL description. Custom periphery and computation logic are mapped to standard cells, bitcell arrays are mapped to "Memory Bricks".

Overall flow. A high-level overview of our LiM synthesis flow is shown in Fig.2. Custom periphery and logic are mapped to compatible standard cells, and bitcell arrays are mapped to "*memory bricks.*" Then using a conventional physical synthesis flow, smart-memory blocks are described in RTL, and physically synthesized using the gate-level netlist. Commercial synthesis tools, such as Synopsys Design Compiler (DC) for logic synthesis, and Synopsys IC Compiler (ICC) or Cadence Encounter for physical synthesis are used to implement the designs. Since all logic and memory arrays are represented at the same abstraction level, any memory block becomes a highly customizable "white box". Libraries of memory bricks and standard cells are used for the LiM synthesis flow. Bricks are integrated by Verilog modules at the RTL, by library files at the gate netlist (.lib that includes timing, power, and area), and as macro blocks at physical synthesis. No modification to the existing physical synthesis tools is required.

Memory bricks. Similar to a standard cell that describes a Boolean logic function, a memory brick is an abstraction for the storage function. As shown in Fig.2, a brick is a bitcell array with simplified local periphery but is not a fully functional memory slice. This simplified structure allows for integration with custom/non-custom periphery, as well as other bricks. They are designed to be stackable so that any banking configuration, memory structure (e.g. different hierarchies and/or partitioning), or memory size that is envisioned in RTL can be implemented. Wordlines (WL) and read/write operations are clocked so that the brick behaves like a sequential cell in the netlist. Decoders and write drivers are not included inside the brick so that they can be synthesized with the logic to allow for smart-memory customization at the RTL level. Any type of bitcell, such as 6T, 8T, CAM (content addressable), embedded DRAM, or multi-ported bitcells can be utilized to form a brick.

Automated brick generation. Brick netlists and layouts are generated by a compiler, and their corresponding library files are generated by a performance estimation tool. This approach enables instantaneous generation of the necessary synthesis files for a given brick. As the memory brick structure is pre-defined for any memory type, we have developed a formulized circuit design methodology based on logical effort calculations and RC delay estimations to automatically size the peripheral blocks within the brick. Taking the memory type, array size (words x bits), and number of bricks to be stacked in a bank as user input parameters, a netlist of a brick is automatically generated by the compiler scripts. Compiled gate sizes are then passed to a layout generator that modifies three main leaf cells (or pre laid-out template cells) of WL driver, local sense, and control block. Leaf cells are pitch-matched to the bitcells, and snap to each other when laid-out in array form. Brick layout is generated by first forming a bitcell array with respect to the user input parameters, and then by arraying the modified leaf cells around the bitcell arrays.

Logical effort [9] is used to optimize the parametric performance of the generated brick. Once the corresponding netlist has been generated, a parameterized library model for the brick is created that includes the critical path, energy, area, and setup & hold times that are needed for use in the subsequent synthesis flow. The gate components within the brick netlist are each represented by look-up table (LUT) models based on bilinear interpolation and curve fitting for delay and energy as a function of fanout and slew rate. Area information and routing blockages are generated by the layout generator and are also part of the brick library model. All input pin capacitances are known from leaf cells. The dynamically generated brick library covers all memory brick sizes, types, and aspect ratios. Any unconventional bit, row, and stacking numbers (non-multiple of 8) are also permitted with such flow.

To validate the accuracy of our estimation tool, two memory bricks with 16x10bits and 32x12bits sizes are generated by the compiler, and their estimated read path delays and read energies are compared to SPICE simulations with RC extracted bitcell array layouts. Several results are summarized in Table-I for different bank sizes of 1x, 4x, and 8x stacked bricks, for reading a word of alternating bits (<1010...10>). For both bricks, error rates are within 2-7% for critical path estimation, within 0-4% for read energy estimation, and 0-2% for write energy estimation.

Table 1. Tool estimation vs SPICE simulation (on RC extracted arrays) for read delay and energy

Number of stacked bricks	Memory Brick 16x10bits			Memory Brick 32x12bits		
Critical path [ps]	Tool	SPICE	Error	Tool	SPICE	Error
1x	247	265	6.6%	295	307	4.0%
4x	269	285	5.5%	322	331	2.6%
8x	292	307	4.9%	353	359	1.8%
Energy [pJ]	Tool	SPICE	Error	Tool	SPICE	Error
1x	0.54	0.54	-0.1%	0.65	0.63	3.1%
4x	0.71	0.70	1.1%	0.88	0.85	3.6%
8x	0.93	0.92	1.3%	1.19	1.16	2.8%

Synthesis. Memory bricks are used as macro cells in the conventional physical synthesis flow, with synthesis files supplied by the dynamically generated brick library. As a simplified synthesis example, the Verilog code in Fig.3 implements a 32x10bit 1R1W SRAM using an 8T bitcell based memory brick with 16x10bit array size (*brick_16_10*). Array size of 32x10bits is first created by instantiating two of 16x10 bricks and stacking them by connecting their input write bitlines (WBL) and output array read bitlines (ARBL). As bricks do not have a decoder, a 5 to 32 decoder generating 32 one-hot decoded wordlines (DWL) is built with standard cells (*decoder_5to32*). Since the SRAM is desired to be 1R1W, the same decoder is instantiated twice for handling read and write addresses. Enable for bricks can be generated from the most significant bit of the address to activate one of the bricks while the other stays idle to preserve energy during reading.

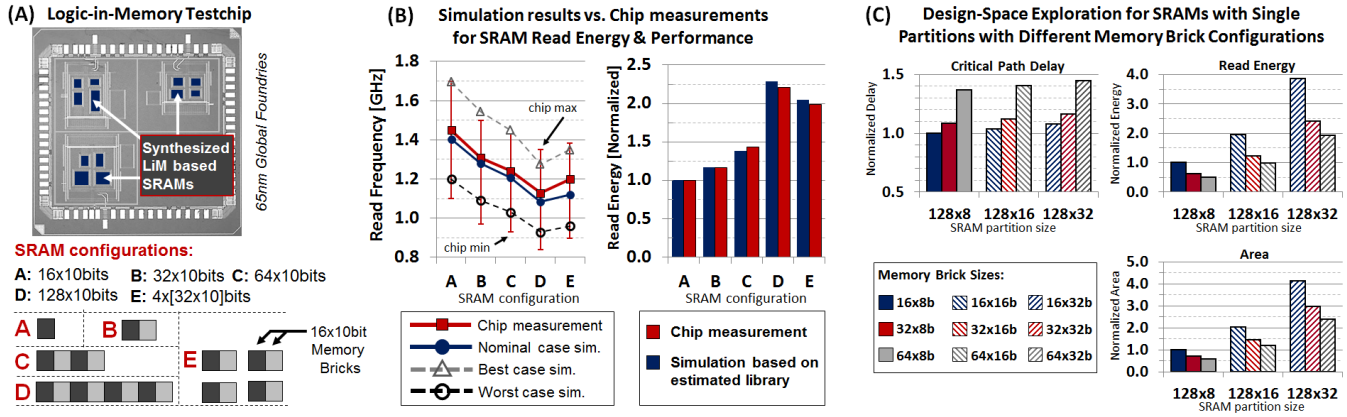

Figure 4. A) Logic-in-Memory (LiM) test chip containing different sizes and configurations of synthesized SRAM blocks. B) Comparison of chip measurements to estimated library based simulations for taped out SRAM configurations. C) Design-space exploration for different SRAMs with single partitions, all synthesized by using different sizes of memory bricks.

```
module SRAM_1R1W_32x10b (
  input  CLK, Read_EN, Write_En,
  input  [4 : 0] RADDR, WADDR,
  input  [9 : 0] DIN,
  output [9 : 0] DOUT            ) ;

wire [31:0] DRWL, DWWL ;

decoder_5to32  R_one_hot  ( .addr(RADDR), .EN(Read_EN), .WL_one_hot(DRWL)  ) ;
decoder_5to32  W_one_hot  ( .addr(WADDR), .EN(Write_EN), .WL_one_hot(DWWL) ) ;
brick_16_10  MEM0 ( .CLK(CLK), .EN(EN_0), .WBL(DIN), .DRWL( DRWL[15: 0] ),
                    .DWWL( DWWL[15: 0] ), .Array_RBL(DOUT)        ) ;
brick_16_10  MEM1 ( .CLK(CLK), .EN(EN_1), .WBL(DIN), .DRWL( DRWL[31: 16] ),
                    .DWWL( DWWL[31: 16] ), .Array_RBL(DOUT)       ) ;
endmodule
```

Figure 3. Simplified Verilog code describing a 32x10bit 1R1W SRAM using two stacked 16x10bit memory bricks.

Accuracy of brick libraries. To validate our automated synthesis flow we implemented a LiM based test-chip that includes different sizes and configurations of synthesized 1R1W SRAMs in a commercial 65nm CMOS technology. Using the same 8T bitcell based memory brick of array size 16x10bits, we implemented different sizes and configurations of SRAMs (Fig.4-a). By stacking the 16x10bit brick for 1x, 2x, 4x, and 8x times to form a single partition, we implemented 1R1W SRAMs with sizes 16x10bits, 32x10bits, 64x10bits, and 128x10bits respectively (configurations *A*, *B*, *C*, and *D*). We also constructed an SRAM of size 128x10bits with 4 partitions (configuration *E*) such that each bank has a size of 32x10bits formed by stacking two 16x10bit bricks. Chip measurements for performance and power are aggregated from multiple chips, under nominal Vdd of 1.2V and room temperature. Simulations on synthesized netlists are done in Synopsys PrimeTime (PT) using standard cell libraries, generated brick libraries, RC parasitic file for routing (.spef), and switching activity file (.saif) that is generated in Modelsim for accuracy.

Comparison of chip measurements and simulations based on the estimated brick libraries are summarized in Fig.4-b for the SRAM configurations. Performance is reported in GHz, and chip measurements are averaged out of multiple chips with maximum and minimum tested speeds shown as bars. Simulation results are shown for best, worst, and nominal cases. Energy numbers are reported for the maximum respected frequencies, and are normalized with respect to the smallest SRAM for ease of analysis. When we analyze both performance and energy, we observe that simulation results are in line with chip measurements and capture the trend of chip results over the range of different configurations within a small error rate. As SRAM size increases for a single partition (from *A* to *D*), performance drops and energy increases as it is expected. For the same size of SRAMs *D* and *E*, partitioning results in faster performance in *E*. Although individual partitions of *E* have the same size with *B, E* is still slower than *B* due to its slower decoder and global signal routing coming from its larger size. Banks of *E* are implemented such that only the bank with the read address hit is activated during read, thus making *E* consume less energy compared to *D*. This gain in energy and performance of *E* however is traded off with larger area consumption that inherently comes from partitioning, when compared to *D*.

These results validate the accuracy of our automatically generated libraries and capture the circuit behavior of the memory bricks efficiently. Error rates are also consistent with the circuit-level results given in Table-I. As system-level simulations based on dynamically generated libraries can capture the trade-offs in between different configurations efficiently, our flow opens up many opportunities for design-space exploration.

Rapid design-space exploration. Enabled by the automated brick generation, we performed rapid design-space exploration to compare various system-level tradeoffs for a simplified case study in Fig.4-c. Three SRAMs with single partitioning of sizes 128x8bits, 128x16bits, and 128x32bits were created. To analyze the impact of brick array size on memory performance, each of the 128xN bit SRAM partitions were built with three different bricks of sizes 16xN bit, 32xN bit, and 64xN bit by stacking them 8x, 4x, and 2x times respectively. Overall, 9 different bricks are compiled with optimum gate sizing (word numbers: 16, 32, 64 and bits: 8, 16, 32).

Performance, energy, and area consumption of these partitions are estimated within seconds by our library generation tool, and the normalized results are summarized in Fig.4-c. As the brick size gets larger, critical path also increases since a brick with larger array size has longer local RBLs. Within the same sized partitions, however, partition with larger bricks consume less energy and area as they have less number of local sense and control blocks per number of words. More interesting results are observed when different memory sizes are cross-analyzed. For instance 128x16bit memory built with 16x16bit bricks is still faster than 128x8bit memory built with 64x8bit bricks, while it consumes nearly the same energy as the 128x32bit memory built with 64x32bit bricks. These results show that array size of the brick and the number of bricks per partition have equally important impact on the overall performance as to overall memory size itself. For this analysis, compiling the netlists and generating the library estimations were finalized within 2 seconds of wall clock time. Thus, the same analysis can be done over a finer resolution of row numbers and bit length without any design cost. Library files for any chosen configuration can be then simply fed into the DC synthesis tool to perform system-level analysis.

To capture the impact of application-specific changes on the system, timing and power analysis are done after logic synthesis. For analyzing the tradeoffs of various memory partitioning and floorplanning choices of bricks, a parameterized Verilog code with parameters describing different memory configurations is used (such as array sizes, banking, memory hierarchy, etc.). For analyzing modifications to the algorithm, the design itself is parameterized by using object oriented tools like "Chip Generator" [13]. Different RTLs can be generated by scripts to explore different corners of the algorithm with varying bus widths, number of parallel cores, or choice of the used arithmetic blocks.

Algorithm and hardware co-design. Building a customized memory provides benefits at the circuit-level, but LiM synthesis facilitates broader benefits at the system-level. At the low-level, all memory and logic are represented at the same level of abstraction, and the white-box memory elements have no hard boundaries. Therefore, all architecture and hardware customizations can be efficiently realized at the RTL. Memory arrays of any size can be integrated with logic in a fine-grained manner, thereby enabling efficient data-streams to be built – even for large datasets. Now that the hardware is customizable beyond what conventional ASIC flow permits, the algorithm constructions can exploit application-level knowledge more aggressively at the high level. With the LiM synthesis flow facilitating the hardware and algorithm co-optimization, superior system-level designs can be realized.

4. LIM SYNTHESIS EXAMPLE

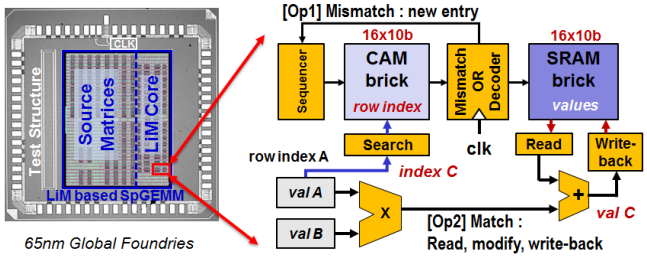

Figure 5. LiM based Generalized Sparse Matrix - Sparse Matrix Multiplication (SpGEMM) CAM architecture.

To demonstrate the efficacy of our LiM synthesis approach we have implemented a data-intensive application in 65nm CMOS technology. The design implementation is a low-power accelerator for Generalized Sparse Matrix - Sparse Matrix Multiplication (SpGEMM), a core function for accelerating graph problems. Graphs are the unified representation of large data structures for modeling networks, data analysis problems, or high-level features of extracted objects in advanced imaging applications. As large graphs are sparse, efficient processing of graphs translates into proper manipulation of sparse matrices. SpGEMM is a core kernel in sparse matrix algorithms such as shortest-path search or graph contraction [4]. However, since sparse matrices have highly unpredictable data access patterns and can be structurally large, SpGEMM operations inevitably cause very high and unpredictable data traffic, leading to serious energy and performance related issues. One way to reduce the data traffic in SpGEMM operations is by using column-by-column multiplication [1], whereby only non-zero elements at the intersections are accessed and processed. Conventional ways to implement this algorithm is a heap based design (priority queue) for computing the columns by using multi-way merging [1], that can be built by first-in first-out (FIFO) based SRAMs. However, FIFO SRAMs cause latency problems due to sequential read/write operations for shifting, which ultimately wastes energy.

To improve the column-by-column algorithm for SpGEMM, Zhu et al. explored the data storage and access patterns in [12] and showed that the SpGEMM operations can be effectively mapped to LiM based content addressable memory (CAM) blocks. As matrix sparsity requires storing only the non-zero elements that are accompanied by their row and column indices, the single cycle "matching" capability of CAMs facilitates index comparison and alignment. Consider a sparse matrix multiplication $C = A \times B$. In this LiM based algorithm, multiplication is separated into two main parts, forming all the columns of resulting matrix C in parallel and assembling them into C. Non-zero elements of a single column of C are formed by using the proposed CAM based architecture for multi-way merging. Row indices of each non-zero element are stored in a CAM array, and their corresponding values are stored in an SRAM array. By using single-cycle CAM matching for cross-checking the intersection of elements in A and B columns, "multiply and add" or "new entry" operation is decided and executed. Since this architecture assembles row indices of each C column, it is called a "horizontal CAM". A similar operation is performed for assembling C by using a single "vertical CAM," which activates individual horizontal CAM blocks only if their corresponding column indices are matched.

High-level system simulations in [12] show that such a LiM based CAM-SpGEMM core can be used as a low-power hardware accelerator in 3D IC stacks. Sparse matrices are decomposed into sub-blocks and then mapped to DRAM rows for maximizing off-chip DRAM row buffer hit. By this approach, access patterns are rendered predictable, thereby maximizing bandwidth of through silicon vias (TSV) for the 3D stack. Sub-blocks of source matrices A and B are stored in the on-chip memory, and the result matrix C is overwritten as it is computed.

We synthesized a CAM based SpGEMM chip in 65nm Global Foundries technology (Fig.5). CAM bricks are compiled with the same circuit formulation as SRAM bricks. Optimum numbers for tile and array sizes for CAM and SRAM bricks are chosen by sweeping array size parameters in gate-netlist simulations on various SpGEMM benchmarks. As a result of this design-space exploration, row index and data array sizes are chosen as 16x10bits, and column number N for sub-blocks is chosen as 32, both consistent with [12]. In contrast to a conventional ASIC flow, since generated CAM and SRAM bricks with such small array size are still very area efficient, the system is synthesized in a fine grained manner from its RTL into 32 horizontal CAMs and 1 vertical CAM with 32 entries. The CAM architecture is shown in Fig.5. For CAM peripherals, a customized mismatch detection block and a sequencer instead of a decoder is built. When there is a match in the CAM, the detection block acts as a priority decoder for the SRAM brick. The SRAM brick is designed as a scratch pad with its customized periphery capable of updating or placing new entries. For updating an SRAM entry, a multiply and add block is integrated with a write-back driver. The resulting LiM based SpGEMM chip area is 1.3mm^2, with a 0.39mm^2 LiM computation core block. A second chip was implemented based on a standard heap based SpGEMM design for comparison. It consumed 1.24mm^2 total area and a 0.33mm^2 computation core block. On-chip SRAM blocks for storing source matrices A and B (Fig.5) are the same in both chips for a fair comparison.

5. SILICON RESULTS

Both the Heap-based and LiM-based implementations were fully synthesized with our approach and the fabricated chips were fully functional. Measurement results in Fig. 6 show that our proposed LiM synthesis methodology provides dramatic system-level benefits for the chosen data-intensive application.

For the same array size of 16x10bits, the CAM brick area is

83% bigger than SRAM brick area, and 26% slower. A single read for the SRAM brick consumes 0.73mW power whereas it is 0.87mW for read and 1.94mW for matching for a CAM brick (based on Spice simulations at 0.8GHz clock). As a result of these circuit-level differences and added customizations in the CAM based architecture, silicon measurements show that the maximum frequency for LiM based SpGEMM chip is 475MHz, whereas it is 725MHz for the non-LiM SpGEMM chip. Furthermore, the LiM computation core block consumes 20% more area.

At the system level we measured the maximum frequencies of the two designs for a nominal Vdd of 1.2V at room temperature. At their respective maximum frequencies, the LiM chip consumes 72mW per clock while the non-LiM based chip consumes 96mW per clock (for both chips, averaged out of multiple test vectors). When these numbers are back-annotated for benchmark sparse matrix operations (University of Florida sparse matrix collection), the LiM based chip offers dramatic energy and performance benefits. Overall latency and energy results for completing SpGEMM benchmarks are summarized in Fig.6. Although the maximum frequency of the LiM chip is 35% slower than the non-LiM chip, the completion time of benchmarks are 7x to 250x faster for LiM chip. Moreover, LiM chip consumes 10x to 310x less energy overall. Utilization of single-cycle CAM matching for multi-way merging drastically reduces the completion time and energy for LiM based SpGEMM chip. Whereas, re-arrangement of FIFO based SRAM arrays at every column computation causes long latency in overall completion time, and higher energy consumption for non-LiM SpGEMM chip.

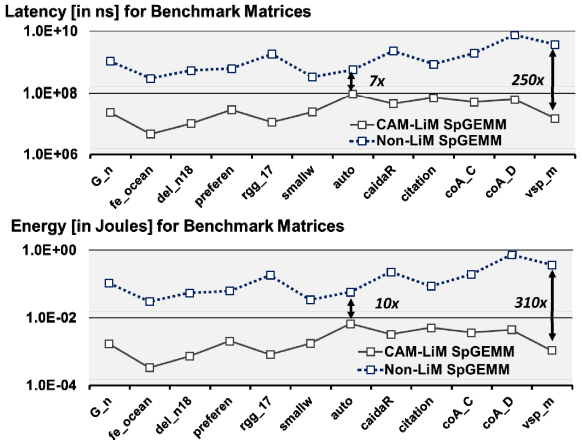

Figure 6. Silicon results of latency and energy for LiM based CAM-SpGEMM and standard non-LiM SpGEMM chips.

6. ANALYSIS AND DISCUSSION

Our results show that superior energy and performance results of the LiM based architecture comes from the co-design of algorithm and hardware by embedding the application specific knowledge into the overall hardware. We have further shown that LiM synthesis flow provides area efficient integration of fine-grained memory arrays based on white-box primitives of logic and memory bitcell bricks. Such logic and memory granularity would be impractical and inefficient with a traditionally compiled embedded memory block approach. Flat synthesis of LiM designs can provide even more area savings when compared to the approach with compiled memory blocks.

The memory brick compiler and performance estimation tools in our proposed flow are technology dependent. Although the underlying circuit methodology and circuit formulas remain the same, technology related characterization of delay-energy LUTs and leaf cells have to be re-implemented when moved to a new technology. This cost, however, is one time only, and vendor supplied memory compilers go through the same iteration as well. Another limitation is since memory bricks are used as macro cells in physical synthesis; synthesis tools do not have the ability to improve the design. One future work for this methodology is to enhance the design flexibility by allowing the selection of memory bricks to be optimized like standard cells. With such an approach, the synthesis tools could optimize the array size and placement of the memory bricks in a standard cell like manner.

7. CONCLUSIONS

As embedded memory consumes more than half of the IC real estate, it is essential to integrate application level knowledge into the memory blocks to improve efficiency for data intensive applications. We have demonstrated that our LiM synthesis approach can enable designers to reliably synthesize application specific block designs using fine-grained “white boxes”. The LiM synthesis flow, by construction, is scalable, manufacturable, customizable, verifiable, and provides efficient co-design of algorithm and hardware as demonstrated by chip implementations.

8. ACKNOWLEDGEMENTS

This work was supported in part by the Intelligence Advanced Research Program Agency and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-12- C-2008. The work was also sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement No. HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government. No official endorsement should be inferred.

9. REFERENCES

[1] Buluc, A., and Gilbert, J.R., On the representation and multiplication of hypersparse matrices, *IEEE IPDPS*, 2008.
[2] Draper, J., et al., The architecture of the DIVA processing in memory chip, *International Conference on Supercomputing*, 2002.
[3] Elliott, D.G., et al., Computational RAM: Implementing Processors in Memory, *IEEE Design & Test of Computers*, 1999.
[4] Kepner, J., and Gilbert, J., *Graph Algorithms in the Language of Linear Algebra*, SIAM, Philadelphia, 2011.
[5] Marinissen, E.J., et al., Challenges in Embedded Memory Design and Test, *Proc. of DATE*, Vol.2, 722–727, March 2005.
[6] Morris, D., et al., Enabling Application-Specific Integrated Circuits on Limited Pattern Constructs, *VLSIT*, 2010.
[7] Murachi, Y., et al., A power-efficient SRAM core architecture with segmentation-free and rectangular accessibility for super-parallel video processing, *IEEE VLSI-DAT*, 2008.
[8] Northrop, G., Design Technology Co-Optimization in Technology Definition for 22nm and Beyond, *VLSIT*, 2011.
[9] Sutherland, I., et al., *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann, San Francisco, 1999.
[10] Vaidyanathan, K., et al., Exploiting sub-20nm CMOS technology challenges to design affordable SoC, *Journal of Micro/Nanolith., MEMS, and MOEMS*, Vol. 14(1), Dec. 2014.
[11] Woo, D.H., et al., An optimized 3D-stacked memory architecture by exploiting excessive high-density TSV bandwidth, *IEEE HPCA*, 2010.
[12] Zhu, Q., et al., Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware, *IEEE HPEC*, 2013.
[13] Zhu, Q., et al., Local Interpolation-based Polar Format SAR: Algorithm, Hardware Implementation and Design Automation, *JSPS*, Vol. 71(3), 297-312, June 2013.