2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors

# Design Automation Framework for Application-Specific Logic-in-Memory Blocks

Qiuling Zhu*, Kaushik Vaidyanathan*, Ofer Shacham†, Mark Horowitz†, Larry Pileggi*, Franz Franchetti*
*Dept. of Electrical and Comp. Eng., Carnegie Mellon University, Pittsburgh, PA, USA
†Dept. of Electrical Eng., Stanford University, Stanford, CA, USA
Email: qiulingz@andrew.cmu.edu, franzf@ece.cmu.edu

***Abstract*—This paper presents a design methodology for hardware synthesis of application-specific logic-in-memory (LiM) blocks. Logic-in-memory designs tightly integrate specialized computation logic with embedded memory, enabling more localized computation, thus save energy consumption. As a demonstration, we present an end-to-end design framework to automatically synthesize an interpolation based logic-in-memory block named interpolation memory, which combines a seed table with simple arithmetic logic to efficiently evaluate functions. In order to support multiple consecutive seed data access that is required in the interpolation operation, we synthesize the physical memory into the novel rectangular-access smart memory blocks. We evaluated a large design space of interpolation memories in sub-20 nm commercial CMOS technology by using the proposed design framework. Furthermore, we implemented a logic-in-memory based computed tomography (CT) medical image reconstruction system and our experimental results show that the logic-in-memory computing method achieves orders of magnitude of energy saving compared with the traditional in-processor computing.**

***Keywords*-Application-specific Logic-in-Memory; Design Automation; Hardware Synthesis; Interpolation; Tomographic Reconstruction;**

## I. Introduction

In the conventional von Neumann model computing systems are physically and logically split between memory and CPUs. This implies that data must be moved between storage and processor. The lower speed of the memory relative to the processor results in the *memory wall*, which has severely limits the efficiency of many applications. When running today's memory intensive applications, modern computers spend most of their time and energy moving data rather than on computation, wasting energy. Further, today's systems are power limited [1], [18], [10], exacerbating the problem.

Recent studies on sub-20nm circuit designs proposed a construct-based design methodology, in which both memory and logic can be mapped together onto a small set of well-characterized regular patterns [12], [13]. This provides opportunities to insert logic patterns reliably next to memory patterns without concern for creating layout hotspots or potential yield problems.

Leveraging this capability we propose application specific logic-in-memory (LiM), as shown in Fig. 1. LiM moves part of a program's computation directly into the memory but keeps the familiar memory interface. It is easy to program, as the computational operations are hidden behind the memory abstraction. In LiM, the embedded logic and its associated data storage are integrated as close as possible, with the primary goal to enable localized computation, thereby minimizing the data transfer between memory and processor and thus saving energy.

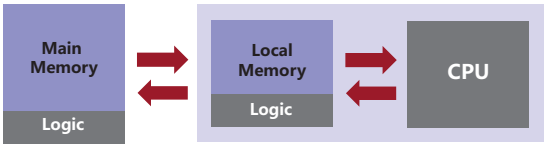


Figure 1. Logic-in-Memory Computing Paradigm: application-specific logic for localized computation is hidden behind a memory abstraction.

A key aspect of LiM is to be application-specific, that is, the embedded logic computation is highly specialized for a particular application domain. A given algorithm class or application domain gives rise to a parameterized design tradeoff space, from which a specialized hardware is then built to meet the required function, under the performance, area and power requirements. LiM benefits from algorithm and application-level knowledge to optimize the embedded logic and memory to a level that is impossible with general purpose computing or configurable hardware computing. Localized computation in combination with application specific logic provides the desired design efficiency of LiM blocks.

A simple example of application-specific LiM is the memory-mapped interpolation, where the interpolated data is accessible as if it was stored in a memory, but actually is computed on the fly from a small seed table. Knowledge about the function to be interpolated (e.g., twiddle factors in the fast Fourier transform, FFT) enables optimization through customization. This simple but versatile LiM block is underlying our discussion of design methods for LiM. The idea of integrating logic and memory is not completely new, but a key challenge is to design such application-specific memory blocks affordably and efficiently.

**Contribution.** In this paper we present a design framework enabling end-to-end hardware generation of application specific LiM blocks that allows users to design, customize, and optimize an interpolation-based LiM structures. We show the applicability for a wide range of applications

1063-6862/12 $26.00 © 2012 IEEE
DOI 10.1109/ASAP.2012.21

from signal processing (e.g., FFT twiddle factor generation) to image processing (e.g., computed tomography (CT) medical image reconstruction). An important observation is that the seemingly simple idea of interpolation becomes a powerful tool when flexible configurability (e.g., relative cost of memory access vs. logic, and different types of logic to embed) is provided in the design phase.

Importantly, the ability to physically synthesis these LiM blocks affordably is enabled by our previous work of "smart" embedded memory synthesis framework which is built from the construct-based logic and memory co-design method [13]. Synthesis results from the framework at 14nm technology show that the resulting LiM hardware is efficient in area, power and latency. Furthermore, our architectural simulations demonstrate orders of magnitude of energy saving from LiM computing compared with traditional processor-centric computing.

**Related work.** The concept of processor-in-memory [10] is grounded in the same observation about the Von Neumann architecture's inherent problem and has been well-studied. Application-specific LiM takes this idea a step further: embedding processors in memory results in too inefficient structures and too much overhead. Thus, application-specific LiM inserts simple, specialized, but powerful logic into memory. Logically similar to our approach, texture mapping on graphics processors [19] provides floating-point addressable memory, where non-integer locations are interpolated. However, the physical implementation is different and the concept of our LiM targets at broader application area.

Interpolation and related areas are well-researched [11]. For example, [9] studied table-based polynomial methods for fast hardware evaluations in FPGAs. In contrast, our work mainly targets at Application-Specific Integrated Circuits (ASICs). The name *interpolation memory* was first proposed by [15] for general continuous function evaluation, and they provide a generalized approach for standard elementary function evaluation that is equivalent to a mathematical library. That work provides detailed numerical analysis of the method. In contrast, the focus of our work is to use the concept of interpolation memory in an application aware manner and to automatically synthesize the resulting hardware on a nanoscale technology.

**Organization.** The paper is organized as follows. In SectionII we describe the concept, design parameterization and tradeoff analysis of an interpolation-embedded LiM block. In SectionIII, we present a tool framework for LiM hardware synthesis. We discuss the results in SectionIV and draw conclusions in SectionV.

## II. LOGIC-IN-MEMORY

In this paper we are developing a general design flow for the application specific LiM. For demonstration purpose, we restrict our examples and evaluation to a special but important class of LiMs, one or multi-dimensional interpolation-embedded memory blocks (i.e., interpolation memory). Such LiMs have a large design space where memory size, interpolation method, and interpolation accuracy are traded off to minimize area and energy consumption under given latency and accuracy. We first detail the general LiM concepts and then discuss the details of interpolation memory.

### A. Application-Specific Logic-in-Memory

**General concept.** Logic-in-memory provides a memory abstraction to move computation to the data, as opposed to moving data to the computing element. From the outside (e.g., a processor) it looks like a regular memory but the structure consists of a smaller memory plus some logic. When reading from an address, the structure returns a value that is computed from the address value and the contents of the memory. A certain part of the address space needs to provide access to the small physical memory to be able to seed the computation. LiM incorporates logic function, or "intelligence," into the memory. While arbitrarily complicated functions are possible, efficiency requires these functions to be simple and only require a few localized memory accesses. LiM is most valuable for power-critical data-dominated applications with irregular and unpredictable memory accesses patterns (thus they cannot exploit the memory hierarchy and caches effectively), and a substantial amount of data transfer will be avoided.

Since the logic is implemented as part of the memory, its functionality must be general enough to benefit enough important applications but also needs to be optimized enough to the targeted application to provide a real gain. It is necessary to pick the right primitives that can successfully be implemented using LiM and are widely enough applicable. Our analysis shows that interpolation memory (discussed next) is such a very general LiM building block that can benefit many signal and image processing algorithms. Further, the high computational demands of these algorithms and the high volume of devices with these applications make it worthwhile to provide specialized and optimized hardware.

**Integration into processors.** From the software perspective LiM acts like a special-purpose on-chip memory or scratchpad, taking in an arbitrary address in the virtual read address space, but returning data that is computed. Further, control and configuration is also exposed through the memory interface. This makes programming the LiM unit very simple. One can easily envision adding a control flag to on-chip storage that engages or disengages the embedded logic, thus not impacting normal execution but significantly enhancing the functionality.

### B. Interpolation Memory

Interpolation memory holds function values at evenly spaced, non-contiguous memory addresses, and the integrated logic performs polynomial interpolation operations on each read reference for locations that do not hold data.

Thus, these LiMs contain a seed table that stores the known function values, and compute "in-between" values on the fly. Interpolation memory has a larger memory read address space than write address space.

**Applications.** Interpolation embedded LiM hardware accelerators can be used for mathematical function evaluation [15], [11] and image transformations (e.g., image rotation, scaling) [20]. In this paper we will particularly discuss the interpolation memory for use in FFT twiddle factor generation and in computed tomography (CT) image reconstruction [6]. In some other situations, the advent of LiM requires to change those algorithms in order to adapt to the logic-in-memory computing. For example, the well-known polar formatting algorithm (PFA) in Synthetic Aperture Radar (SAR) image formation is usually based on a FFT-based polar formatting algorithm, and it is too complicated to be embedded in the memory [5]. In [22], [21], we derive a local interpolation-based variant of SAR PFA algorithm that has equal accuracy as the traditional FFT-based algorithm but fits well in the LiM computing paradigm for energy savings.

**1D interpolation memory.** We first consider the one-dimension (1D) interpolation memory. Assume $N = 2^n$ is the total number of the discrete data points to be evaluated in the evaluation interval. We evenly partition the evaluation interval into $2^k$ segments, so each segment contains $2^r$ elements ($r = n - k$). However, only one element in each segment is stored, so the memory size is reduced from $2^n$ to $2^k$. A polynomial function is defined for each segment to approximate non-stored function values. In this *logic-in-memory* abstraction, $2^n$ is defined as the *local memory size* and $2^k$ is defined as the *physical memory size* and $2^r$ is defined as the *memory compression ratio*. Power-of-2 indexing mechanism is applicable for most interesting problems, and it largely simplifies the hardware implementation.

From the user's point of view, interpolation memory is a full size memory with $2^n$-size address space that supports 1-clock cycle random memory read access. So it has $n$-bits read address space. However, internally the address is split into two parts. The higher $k$ bits are used to address the $2^k$-size physical memory. And the lower $r = n-k$ bits are used to specify the distances between the evaluated data point and its nearest memory references (i.e., interpolation distances). The resulting approximated function value is the weighted approximation of the neighboring memory references, and the weights are set by interpolation distances. The number of nearest neighborhood memory references to be considered is determined by the interpolation order.

**2D interpolation memory.** The idea can be extended to two and more dimensions; we limit the discussion to two dimensions. Given the addresses in the $X$-coordinate and $Y$-coordinate, *2D interpolation memory* returns the corresponding function values in a 2D data array. The returned values are interpolated from its sparsely sampled neighborhood

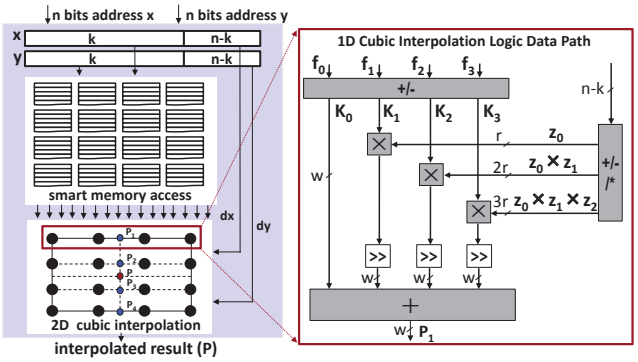


Figure 2. Interpolation Memory Architecture: $n$ bit read address space and $k$ bit write address space (i.e., seed table size); the "in-between" values are approximated from 16 neighboring memory references on the fly.

function values in orthogonal dimensions. These sparsely sampled function values are pre-calculated and stored. In the left part of Fig. 2, we show the hardware structure of a 2D cubic (bi-cubic) interpolation memory.

### C. Interpolation Memory Implementation

We now discuss the hardware implementation of interpolation logic. Using Newton's divided differences, integer data types, and two-power sub-sampling, very efficient implementation is possible.

**Newton's divided differences.** We use Newton's divided differences interpolation polynomial since it is easy to realize in hardware and amenable to be parameterized [3]. The $d^{th}$-order function value $P_d(x)$ is calculated from its neighborhood tabulated functions values $f(x)$ at points of $0, \ldots, x_{(d-1)}$ and it's shown as follows:

$$P_d(x) = k_0 + k_1 \cdot (x - x_0) + \ldots + k_{(d-1)} \cdot (x - x_0) \ldots (x - x_{d-1})$$

For $i \in [0, d-1]$, $k_i = f^{(i)}(x)$ is the $i_{th}$ order divided difference of $f(x)$, and the computation of $k_i$ in hardware for integer data types only involves additions and shifts. $z_i = x - x_i$ are so-called the interpolation distances, which are determined by the lower $r$ bits of the read address. The computational complexity, and the overall hardware cost is proportional to the interpolation order.

**Separable 2D interpolation.** 2D interpolation (e.g., bi-linear, biquadratic, bicubic) can be separated into multiple 1D interpolation in both orthogonal axes. For example, the 2D cubic interpolation in Fig. 2 can be separated into four horizontal 1D cubic interpolations and one vertical 1D cubic interpolation (or vice versa). In the right side of Fig. 2 we illustrate the datapath of a 1D cubic interpolation operation. The bit widths of the data path can also be precisely specified so as not to implement excessive bits, and not to introduce additional error.

**Floating-point interpolation.** For sufficiently smooth functions (which is a precondition for meaningful interpolation), it is easy to convert integer arithmetic designs to full IEEE floating-point designs. Due to the limited dynamic

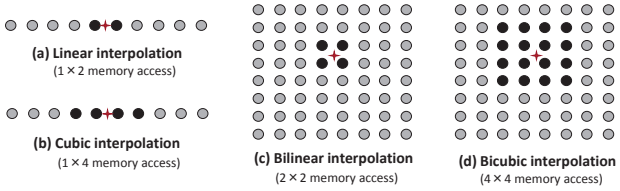


Figure 3. Data Access Pattern in Interpolation: gray array represents the stored function values and the black points are the nearest neighbors to be accessed for the interpolation of the non-stored function values (red stars).

range for a specific known function (e.g., $|\sin(x)| \leq 1$), fixed-point interpolation memory discussed above with some extra precision beyond the floating-point mantissa's precision in combination with some logic and lookup-tables for the exponent identification allows to compose full IEEE-compatible single and double precision numbers with only integer arithmetic.

**Design trade off.** We only consider up to the $3^{rd}$ order polynomial interpolation, that is, linear ($d = 1$), quadratic ($d = 2$), and cubic ($d = 3$). The interpolation order ($d$) together with physical memory size ($2^k$) determine the interpolation accuracy (binary precision bits, $w$). Numerical analysis shows that for any function $f(x)$ that has $d+1$ derivatives, $d+1$ additional precision bits ($w$) of the computed $P(x)$ are obtained for each additional physical address bit ($k$) for interpolating order ($d$) [15]. The tradeoff among these parameters is shown in (1), in which $e$ is the error bits in the precision bits $w$ that are tolerated by the application.

$$(w - e) \propto (d + 1)k \tag{1}$$

(1) gives rise to a design space involving data precision bits, interpolation accuracy, interpolation order, and memory size. And it further leads to different memory/logic area and energy costs for a desired accuracy. In Section III we will discuss our approach to help hardware designers to explore this tradeoff space efficiently and enable them to build the specialized hardware to meet their desired design specs.

**FFT Twiddle factor interpolation.** One important example where interpolation memory can provide a high-precision low-cost alternative is the FFT twiddle factor evaluation. Twiddle factor is the root-of-unity complex multiplicative constant in the butterfly operations of most FFT algorithms. Typically, $N$-point FFT requires equal-size unique discrete $\sin$ and $\cos$ values. Pre-calculation requires $O(N)$ storage while online computation in hardware is expensive [16]. LiM combines small seed table and online computation, thus reduces the overall hardware cost, but still allows single-cycle data access and provides desired accuracy. We will use twiddle factor interpolation memory as an illustrative LiM design example in Section III.

### *D. Rectangular-Access Smart Memory*

Single-cycle interpolation memory requires to access multiple consecutive elements (in 1D or 2D data array) stored

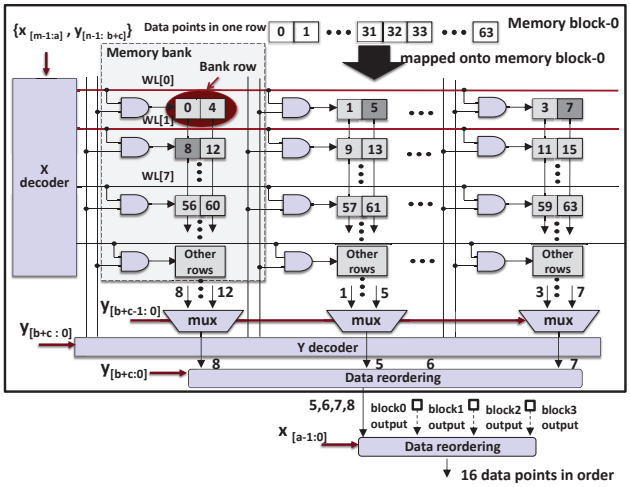


Figure 4. Customized Rectangular Access Memory: customized memory periphery design allows parallel memory banks to share the $x$-decoder.

in SRAM within a single cycle. Traditionally, this is accomplished by distributing data across multiple memory banks so that for any consecutive access all data elements are retrieved from different banks without conflicts. Using multiple SRAM banks incurs high overhead since every memory bank requires its own decoder logic. Using logic-in-memory it is possible to build multi-bank memories that share parts of the decoder logic to exploit the known access pattern. Fig. 3 shows the access patterns for four interpolation memories (linear, cubic, bilinear and bicubic). For example, a $4 \times 4$ rectangular memory access is needed for the bicubic interpolation.

**Basic idea.** We exploit the fact that we always read a constant number of consecutive elements per cycle for each interpolation. The core observation is that after address decoding, the activated wordlines of all memory banks are always adjacent to each other. Based on that, it's possible to optimize the multi-banking memory system to save the periphery overhead. We employ the a customized multi-banking SRAM design topology [14], which provides around $50\%$ area and power savings compared with the traditional multi-banking memory design. However, the design of such customized memory requires careful circuit design, sizing and layout, which is a significant design cost if it cannot be automated.

**Memory design.** We define the functionality of memory to support one-clock-cycle rectangular access of $2^a \times 2^b$ data points from a $2^m \times 2^n$ $2D$ data array. The input of the memory system is the top-left coordinate of the accessing rectangular block $(x_{[m-1:0]}, y_{[n-1:0]})$ and the outputs are all the data point inside the rectangular block. For bicubic interpolation, we have $a = b = 2$ .

To support one-cycle consecutive access of $2^a$ data points in $x$ dimension and $2^b$ data points in $y$ dimension, the parameterized memory is divided into $2^a$ memory blocks; and in each block, there are vertically parallel $2^b$ memory

banks. To control the memory block aspect ratio, we let each word of a memory bank (bank word) holds $2^c$ data points, therefore a block word contains $2^{(b+c)}$ data points. The 2D data array first distributes its $2^m$ data rows into $2^a$ memory blocks row by row (e.g., block-$i$ holds row$[i]$, row$[2^a + i]$, row$[2^{a+1} + i]$, ...). All the $2^a$ memory blocks have the same structure. Fig. 4 shows the organization of block-0 when $m = n = 6$, $a = b = 2$, $c = 2$.

**Localized embedded logic.** The main idea is to let $2^b$ memory banks in each memory block share a modified $X$-decoder by using the same method described in [14]. The $X$-decoder is specifically designed to activate two adjacent wordlines simultaneously. That is, when one block wordline is asserted, the next block wordline is also asserted by the OR gate operation of every two adjacent wordline signals. Another $Y$-decoder is used to select one of the two activated wordlines for each memory bank with the AND operations. Each memory bank word holds $2^c$ data points but each time only one data point of them is required. A column MUX is designed to select one data element for each memory bank and the column MUX is controlled by the lower $(b+c)$ bits of address $y$ $(y_{[b+c-1:0]})$.

As shown in Fig. 4, both the first wordline ($WL[0]$) and the second wordline ($WL[1]$) are initially activated by $X$-decoder but $Y$-decoder further selects the $WL[1]$ for bank0 and $WL[0]$ for the other three banks. After the column MUX, block0 outputs data series of '$8-5-6-7$', which are then reordered to be '$5-6-7-8$'. With some simple logic for data reordering, the smart memory outputs the required $2^a \times 2^b$ data points in order simultaneously. As shown in Fig. 4, the distributions of address bit to each memory component is parameterized. By specify these parameters, the resulting memory architecture can be precisely determined.

When $m = 0$ and $a = 0$, the problem is simplified to $1D$ $1 \times 2^b$ consecutive data access from a $1 \times 2^n$-size $1D$ data array (e.g., $1 \times 4$ neighborhood access in a $1D$ cubic interpolation). The implementation of the memory system is then simplified to have only one memory block.

**Analysis.** Compared with the conventional multi-banking memory design, the amount of memory bank periphery circuits is reduced from $2^{a+b}$ to $2^a$. As is observed in Fig. 4, the resulting memory architecture has the embedded logic gates (e.g. the AND gates) tightly integrated with the memory cells, and each logic gate communicates with its local memory cells. The hardware synthesis of these novel smart memories will be presented in SectionIII.

## III. DESIGN AUTOMATION

Application-specific LiM requires to tailor logic and memory design to application or algorithm specifics. Thus, a strong design automation tool is required to make the approach feasible, as hand-designing of LiMs is prohibitively expensive. We have developed a *design generation and design space exploration tool* for the proposed interpolation memory, which automates the design of its two key components (interpolation logic and rectangular access memory).

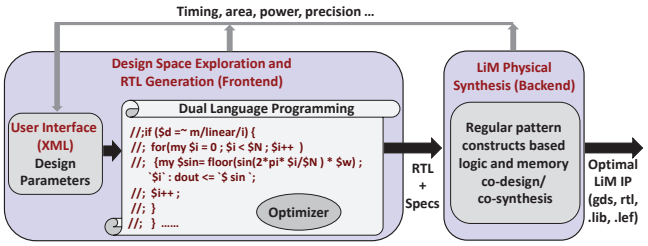


Figure 5. LiM Design Framework

Our tool provides designers with a graphical user interface to select design parameters (e.g., function type, interpolation order, memory size, and precision bits), and generate the corresponding hardware for the specified functionality. Unspecified parameters (free parameters) can be optimized by the system. A designer then evaluates the obtained designs and can explore the design space to optimize the design by varying the parameters. The design framework structure is shown in Fig. 5.

### A. Design Exploration and RTL Generation

**Tool infrastructure.** The tool frontend is built using our chip generator infrastructure "GENESIS" [17], [7] and it's responsible for application interfacing, design optimization and efficient RTL generation. To achieve that, it allows designers to simultaneously code in two interleaved languages: a target language (SystemVerilog) to describe the behavior of hardware and a meta-language (Perl) to decide what hardware to use for given specs. This "dual-language programming" allows to design an entire parameterized family of LiM designs, all at once. Design parameters are set in graphical user interface (GUI) which is defined through XML files. An optimization engine selects optimized values for free parameters. The system supports hierarchical composition of modules and resolving of parameter constraints across modules through all hierarchy levels.

**Illustrative example.** As an example of the application-specific LiM design tool, we show in Fig. 6 the user interface of our FFT twiddle factor interpolation memory design tool. The design parameters are listed in the left panel of Fig. 6. Functional parameters (e.g., *logical memory size*, *precision bits*, *data format*, and *function type*) are set by the application designer or constrained by higher level designs (e.g., FFTs) if used in a hierarchical design. In our example in Fig. 6, the problem is defined of evaluating $128K$ 32bits *fixed-point sin* function values. *Physical memory size* and *interpolation order* are two free parameters that are determined by the optimization engine based on Eq.(1). To achieve the 32-bits precision with the minimum hardware cost, the optimal design point found by the optimization tool is the *cubic* interpolation memory which stores 128 function values. Thus, the tool selects *physical memory size* and *interpolation order* to be 128 and *cubic* automatically. 1D

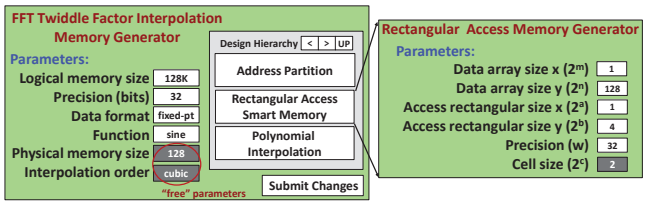


Figure 6. Design Framework User Interface

cubic interpolation needs to access a $1 \times 4$ rectangular access smart memory and this is implemented by the *rectangular access memory design tool*, which is a separate LiM design tool we built and here acts as a sub-module of the twiddle factor interpolation memory design framework. Constrained by the higher-level cubic interpolation memory, it's parameters are shown in right part of Fig. 6. When satisfied with the parameters, the user simply clicks the "Submit Changes" button, the tool will start to run and the dedicated hardware description in Verilog will be generated.

**General framework.** As seen in the example, we are building a LiM tool that is hierarchically composed from lower-level LiM design tools. From these basic building blocks we have built more hardware design tools for larger algorithms, including CT image reconstruction, SAR image formatting, geometric image perspective transformations (e.g., scaling, rotation) and bounded divisions. All of these examples provide users the hierarchical graphical tools to design instances of the algorithm with the capability of exploring the design space to trade off costs and performances.

### B. Physical Synthesis

The automated design framework discussed so far is capable of mapping LiM application specifications to optimal RTL. Our system also relies on a backend "smart memory" compiler to physically co-synthesize logic and memory.

**Smart memory compiler.** Today's embedded memory is typically synthesized using an SRAM compiler. But the use of commercial SRAM hardware IP is unable to incorporate application-specific customization that are required in the LiM design and also hinders comprehensive design space exploration. LiM physical synthesis (the right part of Fig. 5) is enabled by our *smart memory synthesis framework*, which is developed from the pattern construct based logic and memory co-design methodology [12], [13]. Using this framework, embedded logic in the LiM is synthesized together with the memory cells to a small set of pre-characterized layout pattern constructs. Lithographic compliance between the co-designed logic and memory ensures sub-20nm manufacturability of LiM circuits.

**End-to-end LiM design framework.** In our tool chain we are combining the architectural frontend and physical backend to build an end-to-end LiM design framework. Its input is the design specification and the output is ready to use hardware (RTL, GDS, .lib, .lef). When generating a specified design point, our framework also reports the area, power and latency and send them back to the frontend user interface, from which the designer can evaluate the resulting design and reset the design specs for redesign if necessary. Our LiM framework allows an application designer to generate the optimized "silicon" templates by simply tuning the "knobs".

## IV. EVALUATION AND ANALYSIS

We will evaluate the interpolation memory for accuracy, area, latency and energy, for multiple application scenarios. Our automated design framework is used to generate various design points for evaluation. In addition, we built an architectural model for evaluating the energy efficiency of LiM computing paradigm.

**Interpolation memory design tradeoff.** We first investigate design tradeoffs for twiddle factor interpolation memory as we discussed in Section II-C. We built a twiddle factor interpolation memory design tool and generated various design points for 128K-point FFT with data precisions from 6 bits to 53 bits. From Fig. 7(a), the accuracy ($y$ axis) is proportionally increasing with the increasing of the physical memory address bits ($x$ axis) for all the linear/quadratic/cubic interpolation methods. As expected, higher order interpolation method achieves better accuracy for the same physical memory size. To show the relative memory and logic cost of different interpolation methods, in Fig. 7(b), we plotted the physically synthesized memory area and logic area for all the three interpolation methods. The baseline design is the full size memory which stores all the required twiddle factors. All of these designs were set to produce $128K$ integer twiddle factors with 32 bit precision. We see that the total area cost of interpolation memory is much smaller compared with the full size memory. For higher interpolation order, logic area is slightly increased, while memory area reduction is more substantial. So cubic interpolation is considered to be best design choice to minimum hardware cost for a given precision. Design points of 24 bits precision and 53 bits precision can be modified to generate IEEE single and double precision floating-point twiddle factors with negligible additional logic cost.

**Design efficiency of rectangular access memory.** We demonstrate the design efficiency of the rectangular access memory that we described in Section II-D. To be consistent with previous discussions, 1D access windows of $1 \times 2$ and $1 \times 4$ from a $1 \times 128$ data array and 2D access windows of $2 \times 2$, $4 \times 4$ from a $128 \times 128$ 2D data array were studied. For comparison purpose, we also built the traditional multi-banking memory designs of the same functionalities. By using our design framework, we synthesized all the designs with IBM 14nm technology, and measured the power and memory access time for each. We plotted the power-delay product (normalized) in Fig. 7(c) for all the four design pairs. We see that the power-delay-product for rectangular access memory design is about 3 times to 10 times smaller than the traditional multi-banking memory design.

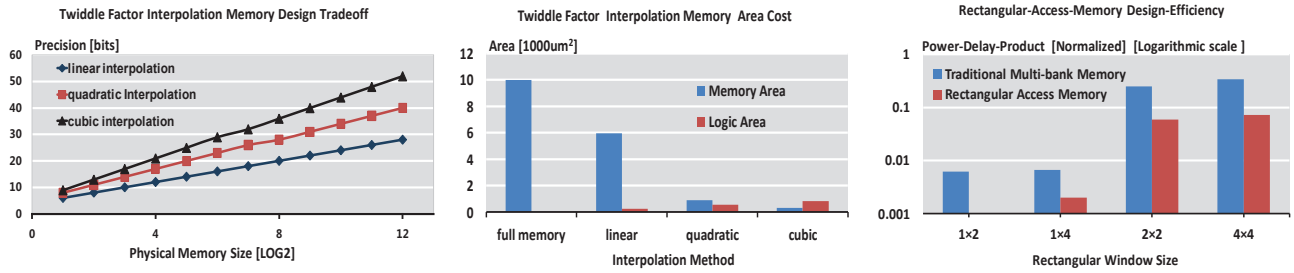


Figure 7. Experimental Results for Interpolation Memory

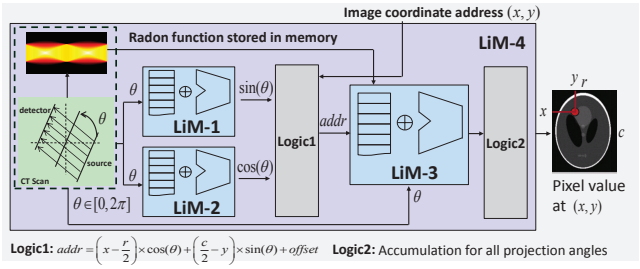


Figure 8. LiM-based CT Imaging Architecture: LiM-1 and LiM-2 implement sin(.) and cos(.) function evaluations; (LiM-3) provides the intrinsic linear interpolation in backprojection. The whole system presents itself as a LiM block (LiM-4), storing the projection data in the memory, but returning the reconstructed image pixels at the querried coordinates.

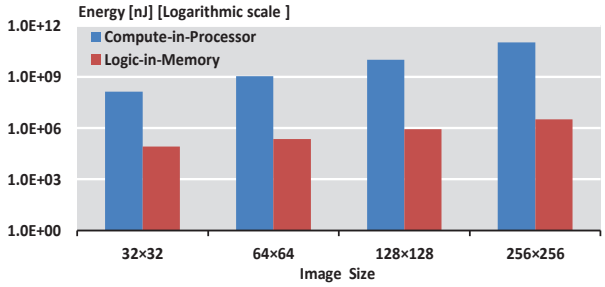


Figure 9. LiM Energy Evaluation for CT Imaging

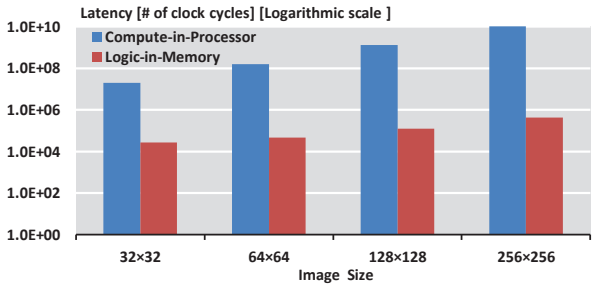


Figure 10. LiM Performance Evaluation for CT Imaging

**Application-level energy savings.** To demonstrate the energy saving of the LiM in a real application, we implemented back-projection-based Computed Tomography (CT) image reconstruction with and without interpolation memory. As shown in Fig. 8, projection data (Radon transform data) from rotation based tomographic scan of an object is obtained and stored in the memory. The inverse of the Radon transform can be used to reconstruct the original image by Shepp and Logan's algorithm [6], [2]. The algorithm involves intensive interpolations operations that can be mapped well to our interpolation memory. As shown in Fig. 8, we implemented the algorithm by using two trigonometric interpolation memory (LiM-1 and LiM-2) for sin and cos function evaluations of the rotation angles where the twiddle factor designs can be reused. In addition, there is another 1D interpolation memory (LiM-3) that is required in the algorithm. It's worth mentioning that the whole system in Fig. 8 presents itself as a LiM block (LiM-4), storing the radon transform data in the memory, but returning the reconstructed image pixel values at the input coordinates.

To evaluate the energy efficiency and performance improvement of LiM computing paradigm, we simulated the CT algorithm in two variants: (1) we performed the logic computation in a simple processor with a direct-mapped SRAM cache storing the Radon transform values, and (2) we used the LiM hardware to store the raw data and also performed the interpolation in the memory, and only sent the final computed pixel value to the processor. Since LiM performs extra logic computation in memory, we scaled its memory accessing power/latency by adding in the normalized logic power/delay from the hardware characterization. We measured the energy consumptions and latency (the number of clock cycles taken to finish the task) for image reconstructions of different image sizes with the Wattch simulator, an architectural level power simulator [4]. As seen in Fig. 9 and Fig. 10, the results show that up to 5 orders of magnitude energy and latency saving were achieved by LiM compared with in-processor computing for the images been studied. The main reason is that LiM eliminates the overhead cost associated with the general-purpose computing, and in addition minimizes the data transferring.

**Other applications and future work.** Application-specific logic enhanced memory customization and its associated suite of design tool present itself as a design methodology that balances cost, performance and energy. The same design methodology can be applied to many other algorithms that have inherent localized and parallel memory accessing pattern. For example, we have built LiM hardware synthesis tools for image perspective transformations, polar-to-rectangular image resampling, and division by a strongly

bounded dividend [8]. Moreover, we exploited the interpolation memory design to build a synthetic aperture radar (SAR) image formation hardware generator, with which we have generated various design points and validated that the interpolation-based approach enables substantial energy savings without sacrificing image quality [22], [21].

Customized to a particular application domain, LiM blocks are particularly suitable for low-power application-specific computing systems like embedded digital signal processing (DSP) processors and mobile graphics processor (GPU). In the future, we will explore LiM design opportunities for such platforms with the goal of enabling energy savings and performance improvements. Our design automation framework is still in the early stages of development and we are working on improving usability and user friendliness.

## V. CONCLUSION

The emergence of constructs based design enables the energy-efficient implementation of application-specific logic-in-memory (LiM). We propose an end-to-end design automation framework for sub-20nm logic-in-memory hardware synthesis and prove its efficacy for an embedded LiM implementing interpolation. We demonstrate the applications of application-specific logic-in-memory blocks in signal and image processing areas, achieving substantial area, performance and energy efficiencies at both architectural and physical level compared with conventional approaches where logic and memory are split in von Neumann architectures.

## ACKNOWLEDGEMENT

The authors acknowledge the support of the C2S2 Focus Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

## REFERENCES

[1] International technology roadmap for semiconductors. 2010.

[2] I. Agi, P. J. Hurst, and K. W. Current. An image processing ic for backprojection and spatial histogramming in a pipelined array. *IEEE Journal of Solid-State Circuits*, 28(3):210–221, 1993.

[3] K. A. Atkinson. *An Introduction to Numerical Analysis*. John Wiley and Sons, 1988.

[4] D. Brooks and V. Tiwari. Wattch: a framework for architectural-level power analysis and optimizations. *Proc.ISCA*, 2000.

[5] W. Carrara, R. Goodman, and R. Majewski. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Artech House, 1995.

[6] C. Chen, Z. Cho, and C. Wang. A fast implementation of the incremental backprojection algorithms for parallel beam geometries. *IEEE Transactions on nuclear science*, 43(6):3328–3334, 1996.

[7] [online] http://genesis2.stanford.edu/mediawiki/index.php.

[8] [online] http://genesis.web.ece.cmu.edu/gui/.

[9] D. J. and de Dinechin F. Floating-point trigonometric functions for fpgas. *Field Programmable Logic and Applications*, 2007.

[10] P. M. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter. Combined DRAM and logic chip for massively parallel systems. *Advanced Research in VLSI*, 1995.

[11] E. Meijering. A chronology of interpolation: From ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, pages 319 – 342, 2002.

[12] D. Morris, V. Rovner, L. Pileggi, A. Strojwas, and K. Vaidyanathan. Enabling application-specific integrated circuits on limited pattern constructs,. *Symp. VLSI Technology*, June 2010.

[13] D. Morris, K. Vaidyanathan, N. Lafferty, K. Lai, L. Liebmann, and L. Pileggi. Design of embedded memory and logic based on pattern constructs,. *Symp. VLSI Technology*, June 2011.

[14] Y. Murachi, T. Kamino, J. Miyakoshi, H. Kawaguchi, and M. Yoshimoto. A power-efficient sram core architecture with segmentation-free and rectangular accessibility for super-parallel video processing. *IEICE Tech. Rep.*, 107(382):47–52, 2007.

[15] A. Noetzel. An interpolating memory unit for function evaluation: Analysis and design. *IEEE Trans. Computers*, 38(3):377–384, 1989.

[16] G. Nordin, P. Milder, J. Hoe, and M. Püschel. Automatic generation of customized discrete fourier transform IPs. *Design Automation Conference (DAC)*, 2005.

[17] O. Shacham. Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms,. *PhD Thesis, Stanford*, 2011.

[18] O. Shacham, O. Azizi, and M. Horowitz. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, 2010.

[19] L. Williams. Pyramidal parametrics. *Computer Graphics*, 17(3), 1983.

[20] G. Wolberg. *Digital Image Warping (Systems)*. IEEE Computer Society Press, 1990.

[21] Q. Zhu, C. R. Bergery, E. L. Turnerz, L. Pileggi, and F. Franchetti. Polar format synthetic aperture radar in energy efficient application-specific logic-in-memory. *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2012.

[22] Q. Zhu, E. L. Turnerz, C. R. Bergery, L. Pileggi, and F. Franchetti. Application-specific logic-in-memory for polar format synthetic aperture radar. *Proc. High Performance Embedded Computing (HPEC)*, 2011.