Design Space Exploration for an Integrated Matrix Inversion Processor with the Gem5-Aladdin Simulator

Mark Blanco and Aayushya Agarwal
Department of ECE
Carnegie Mellon University

I. INTRODUCTION

Over the past two decades, power has increasingly become a primary design constraint in the design of new computer architectures. Improvements in power density from process node shrinkage have ceased to be a reality. As a result, the design of specialized silicon has become nearly commonplace.

These specialized processors often have one of two primary goals: to achieve greater performance and power efficiency in their target workload, or to allow execution of the target workloads at superior energy efficiency, but potentially decreased actual runtime, than current general-purpose hardware. The former is typically referred to as *accelerators*, while the latter, while specialized, are not exactly so.

The amount of computation required by the scientific community has grown with emergent demand for bigger and more complex physics simulations, data processing, and machine learning workloads. Currently many scientific computation workloads are performed on general purpose GPUs, which can be configured to solve algorithms with massive data parallelism. A vital part of several key workloads in optimization and circuit simulation for VLSI designs is the calculation of matrix inverses. This capability enables iterative algorithms such as Newton-Raphson to iteratively arrive at an optimal solution through successive approximations.

II. MOTIVATION

The Newton Raphson method is an algorithm used to find roots of a non-linear equations. This method iteratively solves a linear system of equations using matrix inverse to find the overall solution. Matrix inverse has proven to take a considerable fraction of computation during Newton Raphson. Our study of Newton Raphson indicates that roughly 35% of the execution time is used for matrix inverse. Figure 1 demonstrates the percent of execution time taken for Newton Raphson for sparse matrices with various sizes.

The motivation of this project is to create a heterogeneous architecture that will reduce the time spent on matrix inverse operations. We propose to begin this process with the design of a co-processor that accelerates matrix inverse computation; future work may extend this with

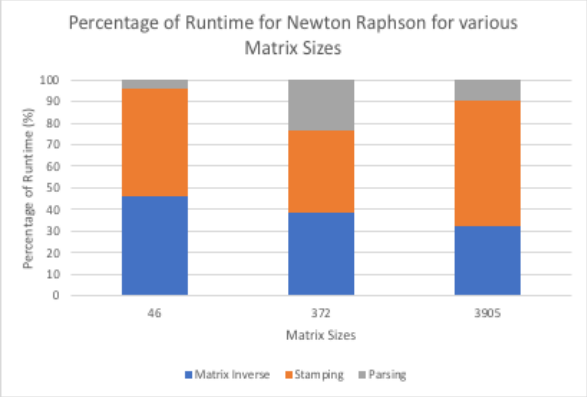

Figure 1 Percentage of Newton Raphson Runtime taken up by Matrix Inverse for various Matrix Sizes

other commonly-used and compute and energy-expensive matrix operations. By exploring the design space for a hardware-parallel matrix inverting processor (MIP), this project aims to increase throughput and lower power consumption and area for scientific computing.

A. Previous Work

Matrix inverse algorithms have previously been parallelized for execution on general purpose Graphics Processing Units (GPGPUs) [1]. This approach implements a dense Gauss-Jordan Elimination algorithm in CUDA and achieves maximal parallelism on a GPU for matrices

whose size is limited such that the overall algorithmic complexity does not exceed the streaming processors (SMs) on the GPU.

Previous works have implemented matrix inversion for typically matrix sizes using Adjoint Method and Gauss-Jordan Elimination [2], [3]. Other implementations focus on application in multiple-input, multiple-output (MIMO) control systems such as multi-antenna radio transmitters, which require high-frequency updates to modulate transmission power [4], [5] and use approximative methods.

In addition to dense algorithms, algorithms that take advantage of sparsity in matrices have existed for some time [6]. These algorithms typically require additional matrix qualities in addition to sparsity, such as symmetry or the banded pattern shown in Figure 2. Therefore, while the circuit simulation problem that is our primary implementation features sparse matrix systems, we do not depend on specific matrix properties in our design space exploration.

$$\begin{bmatrix} B_{11} & B_{12} & 0 & \cdots & \cdots & 0 \\ B_{21} & B_{22} & B_{23} & \ddots & \ddots & \vdots \\ 0 & B_{32} & B_{33} & B_{34} & \ddots & \vdots \\ \vdots & \ddots & B_{43} & B_{44} & B_{45} & 0 \\ \vdots & \ddots & \ddots & B_{54} & B_{55} & B_{56} \\ 0 & \cdots & \cdots & 0 & B_{65} & B_{66} \end{bmatrix}$$

Figure 2 Example of Banded Matrix [7]

III. METHODOLOGY

Our approach for exploring the design space of a MIP begins with selection of the inversion algorithm. Following this, we focus on the integration of the processor with a general-purpose CPU. Then, we briefly discuss the simulation engine used to generate power, area, and runtime estimates for our design sweeps. We then discuss our analysis and evaluation of the design space.

A. Algorithm

As mentioned previously, we considered dense and sparse algorithms for implementation in hardware. Ultimately the straightforward dense matrix inverse algorithm, known as Gauss Jordan Elimination (Figure 3), was selected for its highly parallel structure and general-purpose nature with respect to matrix qualities. The Gauss Jordan Elimination solves for the matrix inverse by stepping through each row, normalizing it with respect to the leading element and then vector subtracting the result from all the other rows.

The hardware and execution complexity of this algorithm grows on the order of $O(n^3)$, where n is the size of one dimension of the matrix. Some optimizations can be made to reduce the compute and memory requirements of the algorithm: matrix columns whose leading values have already been cancelled to 0 in previous main loop iterations can be skipped in later iterations of the algorithm, and the algorithm can be adjusted to provide a direct system solution if a vector b for the system Ax=b is provided in place of an identity matrix of size n. This method does not constrain the matrix structure and can solve for any matrix irrespective of sparsity, even with the above changes.

```
 1 void mat_inv( DATA_T * A, DATA_T * I)
 2 {
 3 #ifdef DMA_MODE
 4      dmaLoad(host_A, dev_A)
 5      dmaLoad(host_I, dev_I)
 6 #endif
 7      main_loop: for i = 0..MAT_SIZE
 8      # normalize the diagonal entry
 9      diag_inv = 1/A[i][i]
10      # Normalize over columns
11      norm_cols: for j =0..MAT_SIZE
12         A[i][j] *= diag_inv
13         I[i][j] *= diag_inv
14
15# Cancel Leading entries of other rows
16      sub_rows: for k =0..MAT_SIZE
17         if ( k == i) continue
18         ref_scale = A[k][i]
19         sub_cols: for j = 0..MAT_SIZE
20           A[k][j] -= ref_scale * A[i][j]
21           I[k][j] -= ref_scale * I[i]j]
22
23 #ifdef DMA_MODE
24      dmaStore(dev_I, host_I)
25 #endif
```

Figure 3 Gauss Jordan Elimination Pseudo-code

B. MIP System Integration

Accelerators can be designed for incorporation into a host system between two extremes. Often, accelerators are designed as large and loosely coupled co-processors that interface with their host processing system through a high-bandwidth data transfer mechanism and use a local memory unit, such as scratchpads, during their computation. This typically serves well for accelerators that exist separately from the CPU's silicon, such as GPUs and some PCIe-connected FPGA or ASIC cards. At the opposite end of the spectrum are tightly-coupled, low-latency accelerators that are either integrated as a functional unit in a CPU pipeline or slightly more distantly integrated on-die with their own L1 cache.

This project considers both memory-offload and cache-based memory interfaces for implementation of a MIP. In the case of the cache-based data interface, the MIP may be integrated directly into an SoC alongside other general-purpose cores. In the case of memory-offload using direct memory access (DMA), the MIP may still be implemented on the same silicon within area constraints. For MIPs designed to handle very large matrices, DMA enables separate implementation on an add-on card.

C. Simulation

This project uses the hybrid Gem5-Aladdin simulation engine to simulate MIP designs that use the aforementioned DMA and scratchpad or cache-integrated data transfer mechanisms [8]. Aladdin is a simulator that performs accurate pre-RTL runtime, area, and power estimation for workloads described in a high-level language such as C [9]. However, Aladdin alone is limited to evaluation of scratchpad only accelerators. Gem5 is a cycle-level CPU simulator that also provides simulation of cache systems [10]. The combination of both simulators allows evaluation of cache connected accelerator designs (Figure 4).

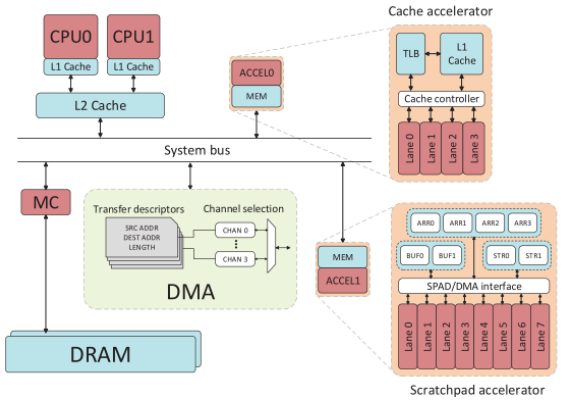

Figure 4: The Gem5-Aladdin Simulator supports Scratchpad and Cache Connected Accelerators. Credit to Shao et al. [8]

D. Design Space Exploration

We evaluate minimization of three objectives for each MIP design. These objectives and their units are as follows:

- Runtime (seconds)
- Average power (mW)
- Total area (mm^2)

Average power and total area each include the contributions of memory (MEM) and functional unit (FU) components. In specific parameter sweeps, we did examine individual contributions of dynamic and static power for MEM and FU as well as MEM and FU area; our Pareto-optimal designs are still evaluated on the three metrics listed above. Pareto-optimal designs are then ranked by their Energy Delay Product (EDP). In most cases Area and Power are highly correlated; thus, we believe EDP is a sufficient summary figure with which to rank jointly optimal designs.

In both scratchpad and cache designs, we pursue a high level of parallelism from the Gaussian-Jordan algorithm by unrolling several key loops. The two for loops, in lines 16 and 19 in Figure 3, known as sub_rows and sub_cols respectively, are points that our accelerator can exploit parallelism. Sub_row is responsible for iterating through all non-leading rows to subtract from the leading row. Within that loop, sub_cols iterates through each column and accesses each value in the matrix to subtract individually. Both loops can be unrolled into several parallel units using the Gem5-Aladdin simulator. To optimize for the best design, it is crucial to find the number of

parallel units for each loop that will meet our objectives. The amount of functional unit parallelism must also be well-matched by the memory system parameters lest over-provisioned compute resources be limited by a deficient memory system.

Our initial parameter sweeps tested several matrix sizes (labeled MAT_SIZE in line 7 of Figure 3) 16, 32, and 64. The maximum matrix size of 64x64 entries, (MAT_SIZE = 64), was selected due to technical limitations; larger simulated scratchpads encountered main memory (RAM) limitations on our evaluation hardware and failed to run. This is the largest matrix size the accelerator can solve without using a decomposition technique such as the Strassen method. Most parameter sweeps used a fixed accelerator cycle time of 5 ns. We show cycle-time scaling of our three objectives for one pareto optimal design in our results.

Following work by Shao et al., we consider several cache and scratchpad parameters. For cache-based designs we vary the overall cache size, cache associativity, cache line size, and number of read-write ports on the cache, termed cache bandwidth in the simulator parameter list. Sweeps for scratchpad designs varied the partition scheme, DMA chunk size, and number of scratchpad ports.

The partition scheme for a scratchpad refers to the type of memory layout and the amount of partitioning that is performed according to this layout. Figure 2 reproduced from Xilinx documentation shows how memory in a flat and contiguous memory layout is rearranged for *block* and *cyclic* partition schemes, both with factors equal to two. At a factor equal to the total size of the original array and for both memory layouts, each entry can be accessed independently by having its own partition. This is *complete* partitioning, which was not considered in our parameter sweeps due for expectation of excessively high area and energy costs and a necessity to limit the range of each design parameter and avoid combinatorial design space blow-up.

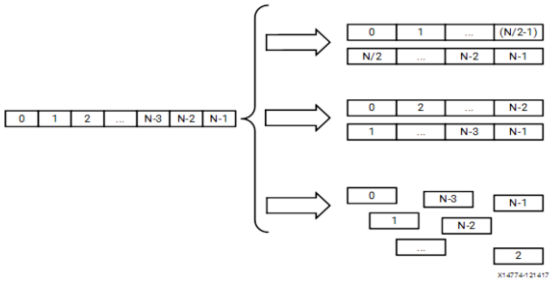

Figure 5 Block Level versus Cyclic Scratchpad Partitioning [11]

IV. RESULTS

Initial design sweeps evaluated all applicable parameters for a given cache or scratchpad MIP, as described in Section III D. Therefore, the ranges of each parameter were limited to 2-3 values to obtain high-level objective trends for high and low parameter values. Further parameter sweeps were subsequently run with the most effective parameters over additional values to generate more detailed information on their' effects on area, power, and performance. Table 1 below lists the parameters swept for cache and scratchpad designs.

Cache	Scratchpad	Both
Cache Size	Part. Type	Matrix Size
Associativity*	Part. Factor	Loop Unrolling
Line Size*	DMA Chunk Size**	Matrix Size
Cache Ports*	SPAD Ports*	Cycle Time

Table 1 Design parameters swept for cache and scratchpad (SPAD) designs.
* These parameters were found to have negligible positive effects on runtime, area, and power.
** These parameters were found to be deprecated towards the end of the project.

A. Effects of Loop Unrolling

In both cache and scratchpad designs, unrolling column-oriented loops (norm_cols, sub_cols) offered larger gains in performance per area and power cost. Unrolling row-oriented loops such as sub_rows had lesser effect on improving benchmark runtime for each matrix size tested.

The effect of row-loop unrolling is magnified for increasing column-loop rolling. This is to be expected as each unrolled version of the row-oriented loop duplicates all unrolled copies of the column-oriented loop contained within. Nonetheless, prioritizing column-loop unrolls over row unrolls is generally more effective because column loop iterations access contiguous sections of memory. Cache-based designs favor accessing contiguous memory segments by prefetching and organizing data in cache lines. Scratchpad designs can be tuned for row, column, or mixed accesses, but generally also favor column access patterns. Figure 6 and Figure 7 illustrate these design tradeoffs.

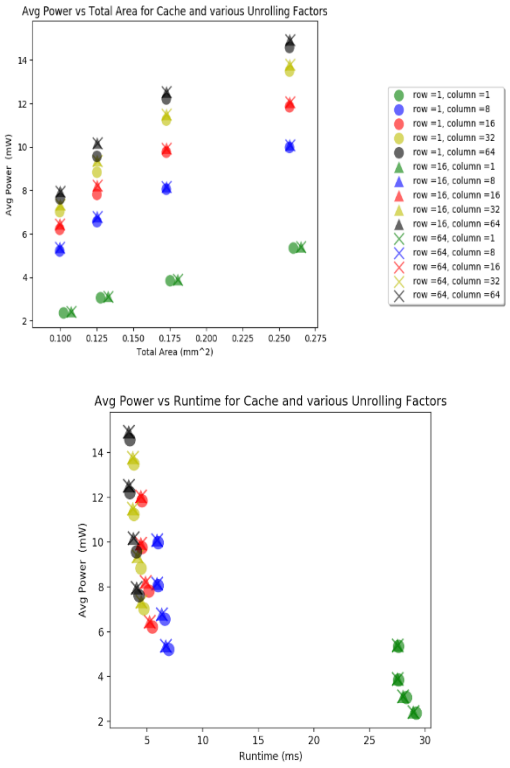

Figure 7 Effects of Column and Row Unrolling on Cache Connected Designs. Points with Identical Marker and Color Correspond to Cache Sizes of 8 KB, 16 KB, 32 KB, and 64KB in order of increasing power and decreasing runtime.

Oddly, area costs were not reflected to increase in row or column parameter sweeps. We are uncertain as to the cause of this anomaly.

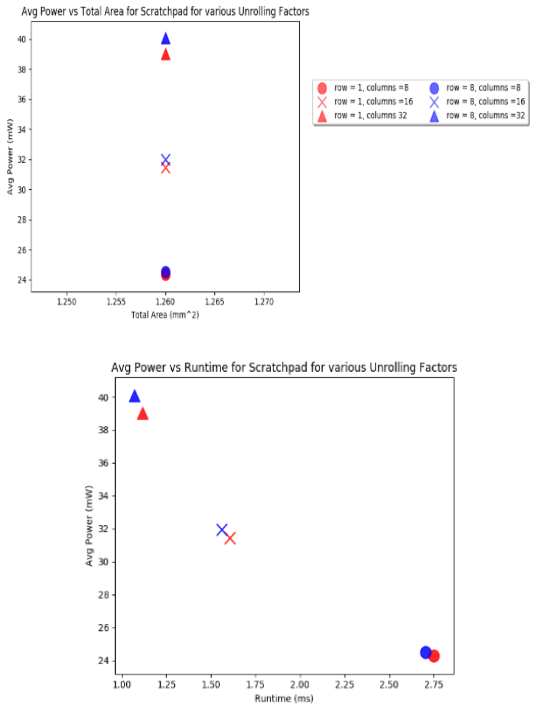

Figure 6 Effects of Loop Unrolling on Scratchpad Designs

B. Cache Parameter Effects

The primary benchmark used for designs sweeps was inversion of a 64x64 entry matrix of float values. Each float value on the test machine was 4 bytes, and the inverse solving algorithm includes both the *A* matrix to be inverted and an equally-sized identity matrix (*I*) in which the final inverted result is computed and eventually returned. Therefore, the memory requirement for an NxN matrix is:

$$Memory\ Size\ =\ NxNx2x4\ Bytes$$

For the 64x64 matrix size this is 32KB of memory. Therefore, our design sweeps covered the range of 8KB (the smallest supported by the simulator) up to 64KB in powers of two. Overall, we find that for design points that have high FU parallelism through high column and row unroll factors, increasing the size of cache substantially increases performance while increasing static and dynamic power costs. The exchange of increased area and power for reduced runtime exhibits limited returns for values greater than 16KB, and

very little gain for 64KB compared to 32KB (Figure 8).

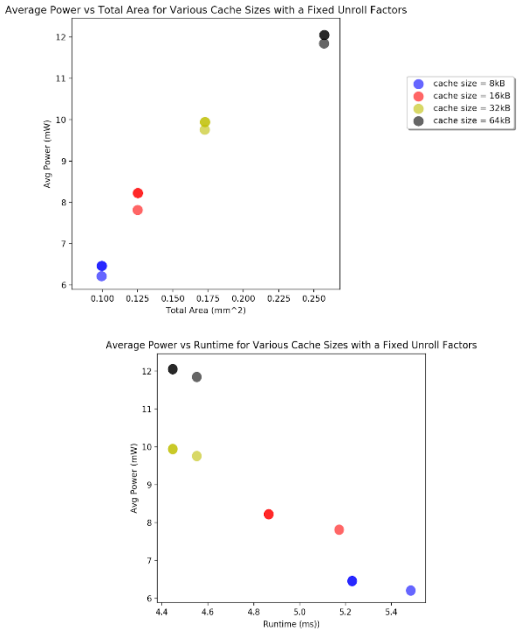

Figure 8 Effects of Cache Size on Runtime of MIP. Larger caches reduce runtime by providing more entries in which to keep still-relevant data used by the functional units.

However, we find that associativity and cache line size had very little effect on the three objectives. Moderate levels of associativity could increase the effectiveness of a smaller cache by adding more entries in which conflicting lines can co-inhabit, but higher associativity increases area and power costs drastically. All Pareto-optimal designs use low values of associativity (2-4). Cache line size was swept for 32 and 64-byte lines; the differences in power and area were minimal but 64-byte lines offered a slight improvement in runtime, so this setting was retained for all later design sweeps.

C. Scratchpad Parameter Effects

In scratchpad designs, the MIP design must be capable of holding the entirety of and array marked for DMA transfer. Therefore, the size of each scratchpad is fixed by the problem size; this leaves partition type and factor as the significant sweeping parameters to influence area, power, and runtime. As referenced in Table 2, cyclic partitioning more readily enables parallel column-level data accesses, while block partitioning at larger numbers of partitions enables independent row-level access (number of partitions equal to matrix MAT_SIZE). Therefore, cyclic and block factors were swept for two separate ranges that each optimized for column-level data parallelism. The ranges are shown in Table 2 below.

Table 2: Number of partitions swept for partition schemes. Note that cyclic factor = 1 was also tested, which equates to one contiguous partition and is equivalent to block with factor = 1.

Block	Cyclic
64	2
128	4
256	8
512	64

Increasing the cyclic partition clearly improves accelerator performance for higher column loop unrolling; however, this comes at the cost of increased area due to scratchpad cost. Power is increased both by the static costs associated with a more complex memory partitioning system and because higher unroll factors are less bandwidth limited by a more partitioned memory (Figure 9). When this is the case, the dynamic power increases considerably as well. Increasing block factors allows larger column loop unrolls to have greater speedup, but also causes a linear increase in memory and increases power.

Analysis of leakage and dynamic power for the memory system and functional units also shows that for a fixed unroll factor, increasing the memory partition factor will allow those allocated hardware units to perform more work, therefore increasing functional unit dynamic power. In mirror fashion, increasing the functional unit parallelism through loop unrolling increases demand on a given memory configuration, thereby increasing the dynamic memory power. When either dynamic memory or functional unit power figures become insensitive to their own factors, this indicates that the bottleneck lies in the other system.

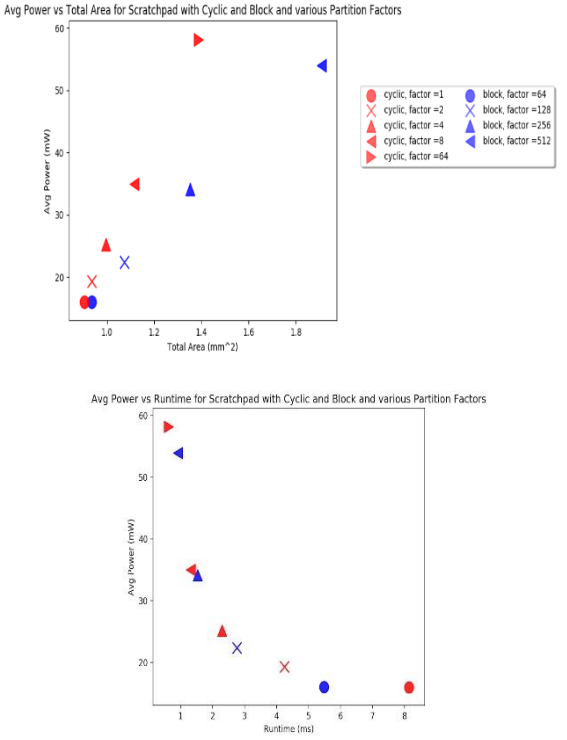

Figure 9 Effects of Scratchpad Partitioning on Highly Unrolled MIP Designs (therefore not constrained by FU, but potentially by MEM).

DMA chunk size as noted in Table 1 was found in a support forum to have been deprecated. Scratchpad ports at the time of this writing does not appear to be a deprecated design option in Gem5-Aladdin, but nonetheless had no effect on any of the design objectives.

D. Matrix Size Scaling

For several designs with fixed loop unrolls (implying fixed hardware size) and fixed cache size where applicable, we swept over problem sizes for 16x16 matrices, 32x32 matrices, and 64x64 matrices. As expected, increasing the problem size has little effect on the power and area of cache-based designs because the cache size is fixed; runtime however scales in accordance with the polynomial computational complexity of the algorithm. Scratchpad designs increase considerably more in power and area costs because the scratchpads must be able to contain the entire working data set, which is on the order of $O(2n^2)$. Because the memory system scales in size (note that partitioning settings were fixed across designs), the scratchpad design can scale to better performance than the cache design, which must pull and evict more and more lines for a larger problem size.

E. Pareto Optimal Design Summary

Figure 10 shows pareto-optimal designs for cache and scratchpad designs. The designs are pareto-optimal over all three objectives and subsequently projected into 2D space; hence there are some designs in each projection that appear pareto-inefficient when viewed for just those two objectives.

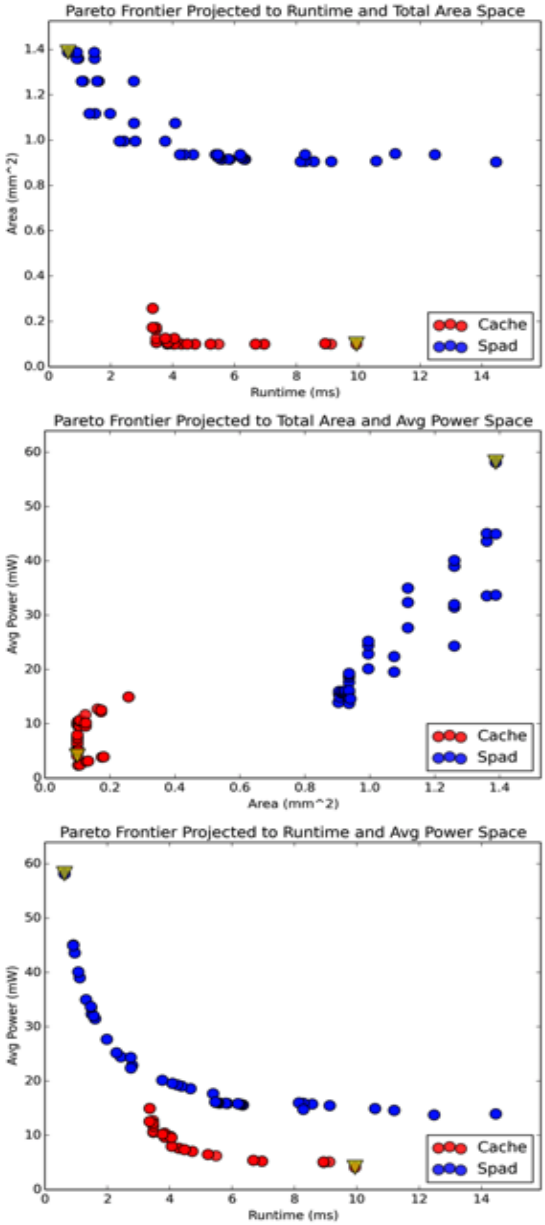

Figure 10 Pareto-Optimal Design Curves for Area, Power and Runtime

It is evident that cache-based designs offer better power and area figures at the cost of runtime. Scratchpad designs offer superior performance at the cost of area and power.

F. Cycle Time Scaling for Selected Design

The design with the second-lowest EDP was selected for further analysis in cycle time scaling. This design point used a cache memory system and had among the best area-energy characteristics of any design, while offering lower runtime than most cache-based designs. This design is indicated by the triangular marker on the red frontier in Figure 10. Scaling cycle times in powers of two from 16 ns down to 1 ns results in predictably decreasing runtime and exponentially increasing power. Oddly, area also increases for designs with faster clock-rates, indicating that Gem5-Aladdin may factor costs of additional clock circuitry to ensure proper timing in a true design (Figure 11).

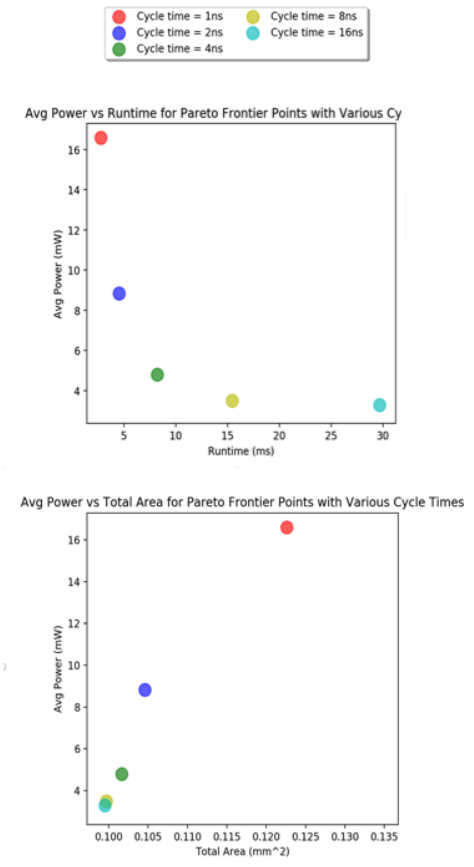

Figure 11 Average Power, Area and Runtime for Pareto Optimal Design for Various Cycle Times

G. Comparison to ARM Processor

We selected a high-performing scratchpad design and tested it at a 1ns clock period. This and the previously selected cache design points are compared against an ARM processor running the dense GJE code as a baseline comparison for area, power, and runtime below (Table 3). In general, the A15 can achieve a much lower runtime than either of the pareto optimal cache or scratchpad designs. However, the pareto optimal designs were able to attain a much lower power and smaller area than the A15 as shown in the last two rows of Table 3. This suggests that our designs can be useful in environments that require lower power and area but can compromise on runtime.

	Power (mW)	Total Area (mm2)	Runtime (ms)
A15 (32nm, 64kB Cache, 2 GHz)	2174	3.29 (3.1 core, 1.9 cache)	0.005914
Cache at 40 nm, 1 GHz	16.59	0.123 (0.075 FU, 0.027 MEM)	2.79
Scratchpad at 40nm, 1 GHz	333.219	1.687 (1.69 FU, 0.107 MEM)	0.135
Cache vs. ARM Comparison	0.76%	3.74%	471.76x
SPAD vs. ARM Comparison	15.33%	51.28%	22.83x

Table 3 Average Power, Total Area and Runtime Comparison of A15 Core versus Pareto Optimal Designs

H. Resource Allocation Between MEM and FU

Energy Delay Product (EDP) was used to rank each globally Pareto-optimal design. This means that all cache and scratchpad designs were considered together and filtered for pareto optimality, which removes some of those seen in Figure 10.

Figure 12 below gives the functional unit (FU) and memory system (MEM) allocations across area, power, and runtime for the globally Pareto-optimal designs. Cache designs consistently achieve the lowest EDP ratings, while scratchpads as a group occupy the high side of the EDP-sorted pareto optimal designs. As illustrated by figures earlier in this section, cache designs use more power and area, especially for functional units, and are thus able to achieve significantly lower runtimes than cache designs. Cache designs can use less memory area, though in fact tend not to reduce their memory power usage because data needs to be fetched, evicted, and re-fetched in each iteration of the algorithm's main loop.

I. Challenges

We encountered several challenges during this project. Initial efforts to sweep large parameter spaces were delayed due to undocumented and breaking issues with the parameter sweeping system included in Gem5-Aladdin.

As such, we wrote our own collection of scripts to automate design sweeps over a flexible set of design parameters. In testing large matrix sizes (128x128 and up), we discovered that the system memory requirements for simulation exceeded out workstation's available RAM, which cause simulations to become unworkably slow when page memory was used, and to crash when that limit was also exceeded.

While online forums were useful to find answers to questions, some details of the simulator went undocumented, such as deprecated design parameters and a bug (which we reported) in which block-partitioned scratchpads completely crashed the simulation at any factor larger than 1.

Finally, we had initially planned to test several algorithms, including sparse algorithms and direct system solving algorithms alongside the dense GJE algorithm. System solving was implemented and tested for small parameter sweeps, but not enough data was collected to make useful conclusions.

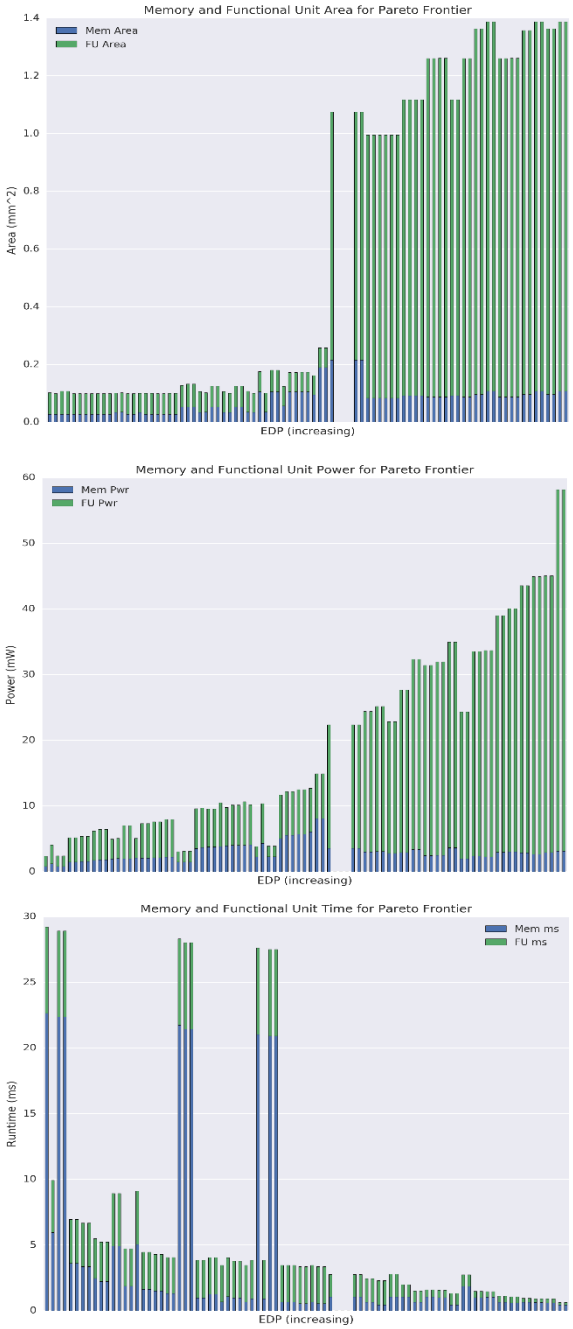

Figure 12 Total Area, Average Power and Runtime Breakdown for Memory and Functional Unit for Pareto Optimal Designs for increasing Energy Delay Products

V. CONCLUSION

Design space parameter sweeps via architectural, cycle-level, or high-level simulation are a useful method to determine the parameters to which design objectives are most responsive. In this work, we explored the design space for a matrix

inverting processor (MIP) using the Gem5-Aladdin hybrid simulator. We investigated the spaces of cache and scratchpad-based memory systems as well as functional unit parallelism through loop unrolling. We find that overall cache-based designs are especially power and area efficient. Scratchpad designs in contrast can scale to lower runtimes with more functional unit parallelism by higher memory system parallelism and repeated memory movement that occurs in a cache system. We compared the area and power of our most efficient and best-performing designs and find that while neither surpasses the runtime of an A-15 ARM processor, we do achieve better area and power figures.

In future work it would be possible to use a mixed cache and scratchpad-based design in which one matrix is accessed via an accelerator-attached cache and the other is loaded and processed upon in scratchpad memory. However, this approach was not pursued in this project. Additionally, general matrix-inverse decomposition methods such as the Strassen Method [1], [12] could be used to enable overflow when a matrix surpasses the hardware capacity of a given MIP. It would also be important to scale the architecture to solve much larger systems. Realistically, a circuit simulation using Newton Raphson is inverting a sparse matrix with hundreds of thousands of rows, much larger than our accelerator could handle. As a result, it will be vital going forward to make the algorithm more robust and capable of handling realistic workload and take advantage of sparsity.

VI. REFERENCES

[1] G. Sharma, A. Agarwala, and B. Bhattacharya, "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA," *Computers & Structures*, vol. 128, pp. 31–37, Nov. 2013.

[2] G. A. Kumar, T. V. Subbareddy, B. M. Reddy, N. Raju, and V. Elamaran, "An approach to design a matrix inversion hardware module using FPGA," in *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, 2014, pp. 87–90.

[3] "Parallel Gauss-Jordan Elimination for Dense Systems."

[4] H. Prabhu, O. Edfors, J. Rodrigues, L. Liu, and F. Rusek, "Hardware efficient approximative matrix inversion for linear pre-coding in massive MIMO," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 1700–1703.

[5] D. Zhu, B. Li, and P. Liang, "On the Matrix Inversion Approximation Based on Neumann Series in Massive MIMO Systems," *arXiv:1503.05241 [cs, math]*, pp. 1763–1769, Jun. 2015.

[6] K. K. Lau, M. J. Kumar, and R. Venkatesh, "Parallel matrix inversion techniques," in *1996 IEEE Second International Conference on Algorithms amp; Architectures for Parallel Processing, 1996. ICAPP 96*, 1996, pp. 515–521.

[7] "Band matrix," *Wikipedia*. 28-Apr-2017.

[8] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[9] Y. S. S. Brandon and R. G.-Y. W. David, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," p. 12.

[10] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] "Increasing Local Memory Bandwidth." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/aio1517252127912.html. [Accessed: 06-May-2018].

[12] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, Aug. 1969.