SPARK: Sparsity Aware, Low Area, Energy-Efficient, Near-memory Architecture for Accelerating Linear Programming Problems

Siddhartha Raman Sundara Raman, Lizy John, and Jaydeep P. Kulkarni
The University of Texas at Austin
s.siddhartharaman@utexas.edu

***Abstract*—Integer Linear Programming (ILP) is an important mathematical approach for solving time-sensitive real-life optimization problems, including network routing, map routing, traffic scheduling, etc. However, the algorithms for solving ILPs are typically sparse and branch-intensive, and not CPU/GPU friendly. In the paper** *"What could a million cores do to solve Integer programs",* **Koch et al. [40] presented data illustrating that Integer Linear Programming (ILP) applications take tens of hours of execution time even on the largest parallel computers. Long execution time is a problem because many real-life applications need a decision in seconds or minutes. The widely used ILP solvers, like Gurobi (optimized for CPUs), perform software-based optimizations to handle the inherent sparsity in ILPs but still do not meet decision threshold because of the limited throughput of CPUs. GPUs are suited for large-sized dot-product compute, however, GPU-based ILP solvers also do not meet decision thresholds as (i) GPU is not sparsity friendly and (ii) GPU incurs thread divergence for branching, resulting in under-utilization of streaming engines and periodic host-GPU interaction. We propose** *SPARK, a sparsity-aware, reuse-aware, energy-efficient, reconfigurable, near-cache ILP architecture* **that (i) re-configures the existing L1 cache present in CPUs to perform near-cache acceleration with easy integration into the baseline CPU pipeline with minimal area overhead ($\sim$ 1.4% of a CPU), (ii) performs near-cache sparsity detection and sparsity-aware compute, reducing the number of insignificant computations, and data movement energy overheads, (iii) leverages the computational patterns present in algorithms used for solving ILP to realize a reuse-aware architecture, and (iv) is applicable to solving sparse and dense ILPs and LPs (Linear Programs). We observe 15x/20x, and 152x/740x performance/energy improvement over AMD's Zen3 CPU, and Nvidia's Tesla v100 GPU for sparse real-life ILPs in Mixed Integer Programming library (MIPLIB 2017). For sparse LPs (non-integer), SPARK achieves 7-17x/103-250x performance/energy improvement over CPU/ GPU indicating SPARK's broad applicability.**

I. Introduction

Linear Programming (LP) [14] [12] is an essential mathematical tool used to analyze a variety of optimization or feasibility problems, with the solution to these problems deduced based on a set of linear constraints. A variant of linear programming called integer linear programming (ILP), adds complexity by restricting the solution space to a set of integers. Solving ILP problems has been historically challenging [38], [40]. Firstly, real-world ILP applications, like network routing, stock investments, traveling salesman, and emergency dispatch scheduling [38], demand timely decisions. Optimizers like Gurobi [23] and CPLEX [13] use data patterns and multi-threading on CPUs for precise solutions. Koch et al. [40] show many ILP executions take tens of hours, even on 4000-8000 cores. While GPUs are an option, dataset sparsity (65-99%) poses a challenge [21]. Execution times of state-of-the-art optimizations on CPUs and GPUs, as shown in Fig. 1 for selected applications from MIPLIB 2017 [20], significantly surpass the decision threshold time for time-critical applications, even when leveraging libraries like cuSparse for GPU-based sparse problem execution. **Secondly**, the energy to converge at the solution, can be extremely high ($\sim 10^6$ Joules), when MIPLIB benchmarks are executed on a CPU, because of data movement overhead and computational costs with large-sized sparse ILPs.

MIPLIB 2017/ 2010 instance	Time-sensitive Real-life application	CPU+Gurobi solution time	GPU+cuSparse solution time	Decision threshold
Ns1111636 (NS)	Network routing	103 hrs	105 hrs	<10mins
Markshare2 (MS)	Market sharing	1.5 hrs	1.75 hrs	<1min
Stp3d (ST)	Map routing	114 hrs	110 hrs	<1min
Timtab1 (TT)	Traffic scheduling	10 mins	8 mins	<30 secs
Air05 (AR)	Airline scheduling	45 mins	40 mins	<5mins
Blpar98 (BL)	Railway planning	30 mins	35 mins	<5mins
gen-ip054 (GE)	Random ILPs	1.25 hrs	1.7 hrs	<5mins

Fig. 1. **ILPs on CPUs and GPUs do not converge at the solution within the decision threshold time for time-sensitive real-life applications.**

Domain-specific accelerators show promise for large problems with high runtimes and energy demands [35] [36] [31] [64], but solutions without dedicated accelerators are preferable. While the dot product in ILP benefits from parallelism, sparsity and control-intensive tasks present challenges. Additionally, moving large data between the host CPU and accelerator is a challenge. Our solution involves a near-memory architecture integrated into the CPU, leveraging L1 cache for compute and minimizing data movement overheads. We allocate small area for dedicated sparsity-aware peripheral logic near L1 cache to handle control-intensive parts of ILP. The major features of SPARK are:

- **Sparsity awareness:** A sparsity-aware algorithm is proposed to alleviate insignificant computes along with the ability to perform useful compute leveraging the high throughput of PIM, leading to energy efficient compute.
- **Reconfiguring CPU components:** Spark is tightly integrated into the CPU pipeline, and reuses existing CPU components such as L1 cache, to accelerate ILP.
- **Reuse-awareness:** Identification of computational pat-

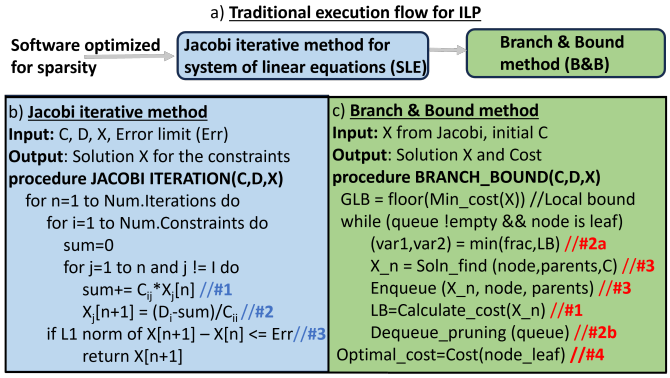


Fig. 2. **a) CPU/GPU-based ILP execution involves software optimization of sparsity, followed by SLE, B&B methods. Pseudocodes for b) Jacobi iterative method (SLE) and c) Branch and Bound (B&B).**

terns in ILP algorithms to enable reuse of near-memory logic across different compute engines.

- **Near-cache architecture:** The fine-grain near-memory dot product compute maximizes parallelism, while the coarse-grain dedicated hardware ensures high performance for control-intensive tasks, minimizing data movement through in/near-memory computing.
- **ILP Speedup**: Spark achieves 15x/20x performance, and 152x/740x energy improvements over AMD's Zen3 CPU/Nvidia's Tesla v100 GPU for sparse ILPs.
- **Broad applicability - LP Speedup**: Spark is also suitable for LP problems in addition to ILP problems, and results in 5-7x/150-180x performance/energy improvement over CPU/GPU in dense LP, and 7-17x/103-250x performance/energy improvement over CPU/GPU in sparse LP.

II. BACKGROUND

A. Integer linear programming (ILP) formulation

Linear programming [19] (LP) solves optimization problems with non-negative solutions. Integer linear programming (ILP) [4], [39] (special case of LP) restricts solutions to integers [7], leading to exponential time complexity.

ILP constraints are represented by (i) a 2D matrix (C) with rows/columns equal to the number of constraints/variables, and (ii) a vector (D) with rows equal to the number of constraints. Additional constraints like $X_j \geq 0$ and $X_j \in Z$ restrict the solution to non-negative integers [42], [44].

Thus, the general form of **optimization** version of ILP problem is $OptimizeF(X) = \sum_{j=1}^{N} A_j * X_j$. The general form of the feasibility version of ILP problem is B = $\sum_{j=1}^{N} A_j * X_j$, where N is the number of variables, represented as $X_{1\text{-}N}$ in a system of linear equations. ILP's solutions satisfy these constraints: $i) C * X \leq D; ii) X_j \geq 0; iii) X_j \in Z$

B. Traditional execution flow for solving ILPs

Direct algorithms [18], [32], [37] excel with smaller constraint sets, while iterative ones handle real-time optimization with complex constraints. Iterative methods like Jacobi and Gauss-Seidel [2], [49], [59] find optimal solutions, relaxing integer constraints, and Branch and Bound (B&B) [17], [61] refines these to integer solutions. Fig. 2.a shows ILP flow, with sparsity-optimized software executing SLE followed by B&B. MIPLIB 2017 benchmarks reveal sparsity levels of 65%-99%.

C. Jacobi method for system of linear equations (SLE)

This iterative numerical technique solves a system of linear equations. It works with input constraints (matrix C), variables (X), and constants (D). i) **Initial Approximation**: Begin with a randomly initialized solution vector X. ii) **Iterative Updates**: Refine the solution iteratively by updating each variable. Each iteration computes new variable values based on the old values of all others. In Fig. 2.b, #1 and #2 find updated X using MAC, subtraction, and division. iii) **Convergence criterion**: The process continues until a convergence criterion is met, which occurs when changes in X between iterations become sufficiently small (L1 norm). Label #3 in Fig. 2.b checks if L1 norm is less than a predefined error limit.

D. Branch and Bound (B&B)

B&B, after Jacobi, finds the optimal integer solution by generating child nodes for each parent node and selecting a branching variable based on the node's established bounds. B&B for minimization (min) problem involves steps shown in Fig. 2.c: i) **Branching start**: The lower bound (LB) is initialized to the ceil of $\sum_{j=1}^{n} A_j * X_j$. Global UB is the value of F(X) when X is the ceil of the obtained solution from SLE (#1 in Fig. 2.c); ii) **Branching node/variable**: Variable/node with the least fractional/local LB part branches first (#2a in Fig. 2.c); iii) **Branching continuation**: C matrix updates each iteration to include new constraints from B&B, and the updated X matrix is identified at each branch with the local LB of each node (#3 in Fig. 2.c). iv) **Branching complete**: Pruning occurs in four ways: (a) The solution vector X from SLE consists of non-negative integers. (b) When the local LB equals the global UB, and other local LBs are less than the global UB. (c) F(X) is identical across leaf nodes, with at least one node having non-negative integers. (d) F(X) is the same across some branches, with at least one valid solution and others infeasible (#2b in Fig. 2.c).

III. MOTIVATION

A. Understanding 3C criterion for ILP execution

To explain SPARK's necessity, we highlight why existing accelerators are unsuitable for ILP execution. ILP involves more than just matrix operations, requiring the **3C criterion**: **C1** - Dot-product intensive operations in SLE, **C2** - Handling sparse constraints, and **C3** - Managing control flow tasks like L1 norm in SLE, B&B. Existing accelerators typically fulfill at most two of these, not all.

B. Shortcomings of Linear Algebra/Tensor accelerators

Traditional linear algebra accelerators handle dot-product intensive operations such as matrix-matrix/matrix-vector multiplications in convolution, fast-fourier transform for signal processing. These are mainly classified as application-specific

MIPLIB 2017/2010 instance	GPU+cuBLAS solution time	CloudTPU solution time*	CGRA solution time*	Estimated SparseTPU*
Ns1111636 (NS)	108 hrs	244 hrs	250 hrs	150hrs
Markshare2 (MS)	3.5 hrs	9hrs	19 hrs	6.5hrs
Stp3d (ST)	114 hrs	272 hrs	300hrs	195hrs
Timtab1 (TT)	9.5 mins	23mins	1hr	20mins

Fig. 3. **Experiments with TPUs/CGRAs show unacceptable solution times (in hours), even at reduced accuracies (* indicates 98% of CPU accuracy is achieved).**

integrated circuits (ASIC) like TPU, Bison-e, and reconfigurable architectures like CGRA, Transmuter [46], etc.

Experiments with TPUs for ILP workloads show they underperform compared to CPUs/GPUs, which already struggle to meet decision thresholds (Fig.1). Results in Fig.3 align with CloudTPU documentation [1], highlighting TPUs' inefficiency for branching and sparse operations. Additionally, solution accuracy on TPUs is only 98% of that on CPUs, compromising both accuracy and solution time.

Bison-e [48], an ASIC optimized for generic integer linear algebra applications, uses binary segmentation for matrix-matrix/vector multiplication, addressing characteristic C1. However, Bison-e currently lacks control for handling sparse, control-flow intensive operations, unsuitable for ILP, and framework/compiler support is still in nascent phase.

Coarse-grained reconfigurable architectures (CGRA) are valued for their reconfigurability, mapping problems to data-dependency graphs (DDG). However, they rely heavily on compilers and perform poorly on ILPs with conditional statements [15]. Data sharing between the CPU and CGRA is complex with current frameworks, limiting their suitability for real-world ILPs (Fig.3). We also observed sparsity and control flow issues causing underutilization of processing engines.

Transmuter [46] uses a reconfigurable data-flow model (like CGRA), adaptable memory, and cross-bar arrays, allowing kernel computation with varying arithmetic intensity. However, it has drawbacks: i) Only 2x speedup over CGRA, insufficient compared to CPUs/GPUs. ii) Frequent data movement from the host CPU. iii) High integration overhead for control engines resembling local CPUs. iv) Unclear energy cost for clocking local control units. v) High reconfigurability overhead for sparse/branching workloads.

C. Shortcomings of existing sparsity-aware accelerators

Numerous sparsity-aware accelerators, dedicated to machine learning, are unsuitable for solving ILPs due to their i) inability to handle control-flow-intensive operations. ii) Offloading such operations to the host requires periodic host-accelerator data movement, impacting performance/energy.

Sparse-TPU [28] addresses sparsity by leveraging static sparsity in weights, allowing offline encoding of data and transformations for arranging them as structured dense computations. i) Sparsity-aware TPUs incur a fundamental overhead. Data must be pre-processed to ensure that non-zero elements access the correct index/PE block across processing engines aligned in a mesh style, adding extra overhead. [63] ii) The proposed 2D tensor matrix approach requires adaptation to

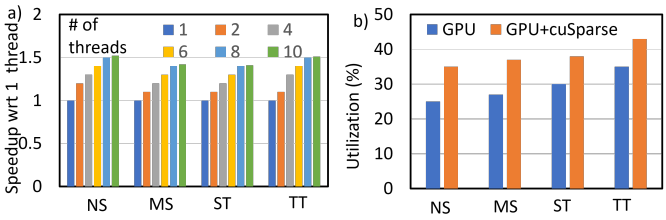


Fig. 4. **a) ILP on CPU - Performance saturation with increasing number of threads suggests hardware bottlenecks like limited throughput and high data movement. b) ILP on GPU - GPU utilization with/without cuSparse is less due to sparsity and thread divergence**

modern TPUs at a 3D level. Column packaging may not achieve optimal density for a 3D matrix. iii) Software-based sparsity encoding/detection is slower compared to extremely parallel hardware-based sparsity detection. iv) Sparse TPUs, at best, exhibit characteristics of C1 and C2, but struggle with control-flow-intensive operations, leading to frequent data movement cost and rendering them unsuitable for ILP operations. v) A reported 16x speedup aids SLE but not L1 norm or B&B, falling short of CPU/GPU performance levels. Impact on solution accuracy remains uncertain.

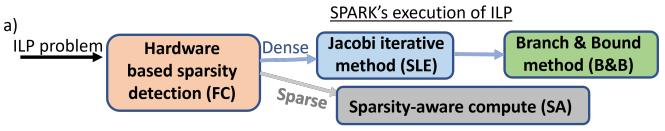
EIE [24] is a specialized DNN hardware accelerator that employs Deep Compression for network pruning and utilizes a dedicated pipeline for matrix-vector multiplication. Other sparsity-aware accelerators, such as SparTen [22] and ExTensor [29], require components like prefix-sum adders and content address match, designed specifically for CNN. These cannot handle B&B or L1-norm operations due to their control logic limitations, requiring offloading to the host CPU, increasing data movement and energy costs. To modify them into ILP accelerators, additional structures like queues, subtractors, and dividers would be needed for B&B and L1-norm, effectively requiring a separate accelerator. This is because they are designed only for sparse dot-product computations, and their existing structures can't be reused for B&B/L1-norm.

D. Shortcomings of Ising/Boltzmann accelerators

Boltzmann/Ising accelerators map NP-complete Combinatorial Optimization Problems onto an Ising graph, where spins (S) and interaction coefficients (IC) represent variables and constants, respectively. Existing Ising accelerators face limitations for ILPs: i) Binary-valued spins limit their applicability to binary ILPs, unlike real-life applications. ii) Most Ising accelerators like [62] [52] [55] rely on Hamiltonian energy being represented as a quadratic formulation of product between S and IC, suitable for PIM compute. However, binary ILPs warrant a different Hamiltonian energy (mentioned in [43]), not captured by the existing Ising accelerators.

E. Shortcomings of CPU/GPU based execution

While multicore CPUs can be used for ILP solving, Koch et al. [40] presented the inadequacy of solving ILPs on multicores, Fig. 4.a suggests that increasing threads do not scale performance ([40]). CPUs, using sparsity-optimized software like Gurobi rely on Von Neumann compute.



b)

Problem	Proposed solution
Accelerate Jacobi iterative method (SLE)	In-L1 cache dot product/near-L1 cache arithmetic operations
Accelerate Branch & Bound method (B&B)	Reuse-aware in/near-L1 dot product/arithmetic operations
Hardware based Sparsity detection (FC)	Near-L1 sparsity detection
Sparsity-awareness (SA)	Near-L1 sparsity-aware compute
Dead silicon area in accelerators	Tightly integrated with CPU's L1-cache

Fig. 5. **a) SPARK uses hardware-based sparsity detection, execution based on sparsity. b) Problems in solving ILPs and proposed solutions**

GPUs can solve ILPs, but the sparsity poses challenges [21]. Data is transferred from the CPU's Data (D) cache via shared memory to GPU cores, incurring data movement overhead leading to energy bottleneck. Fig. 4.b shows the under-utilization in GPUs with/without cuSparse, because of sparsity and thread divergence [23] [10], negatively affecting efficiency [25] [33]. GPU+cuSparse/cuBLAS (Fig.4) were compared, with the former outperforming the latter. Subsequently, results from cuSparse alone are presented.

F. Challenges in existing B&B accelerators

Software optimizations to accelerate B&B have been proposed [10], [50], while no hardware accelerators for B&B exist. To reduce thread divergence in GPU-based B&B, Chakroun et al. [10] used an entirely software-driven optimization for executing branches in parallel. The authors of [50] propose a fast algorithm for optimal sub-problem identification in feature selection. However, these suffer from sparsity, necessitating periodic host-GPU interaction and unnecessarily lengthening the time of short threads.

G. Challenges in existing Linear Programming accelerators

Prior research [9] [8] investigates the use of a linear solver System on Chip (SoC) architecture that utilizes a Residue Number System (RNS) combined with residual processors (RP) functioning as SIMD (Single Instruction, Multiple Data) units. While this approach offers parallel computation benefits, RNS suffers from significant limitations, primarily its low dynamic range, which restricts its ability to handle problems with a wide range of values. This limitation hinders convergence, reduces performance, and makes it difficult to apply the system to problems with more than 10-20 constraints or larger-scale Linear Programming (LP) problems. Another study [6] explores the use of FPGA for implementing the Simplex algorithm, leveraging block-RAM for data storage to speed up the optimization process. However, this FPGA-based solution faces scalability issues, particularly in terms of its ALUs, and it lacks the necessary support for solving Integer Linear Programming (ILP) problems using Branch-and-Bound (B&B) methods.

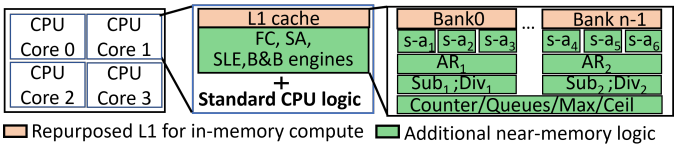


Fig. 6. **Spark is realized by re-configuring L1 cache (orange) in CPUs for PIM along with minimal near-memory logic (green) shared among FC, SA, SLE, and B&B engines. In an L1 cache with n banks, engines are realized using: shift-add ($s\text{-}a_{1\text{-}3}$) at a finer granularity of 1 per 16 columns in a bank for 16-bit compute, adder reduction (AR_1) for s-a outputs, subtraction (Sub_1), and division (Div_1) at a coarser granularity of 1 per bank. The counters/queues/max/ceil are shared across the cache.**

H. Challenges in existing processing-in-memory accelerators

PIM accelerators have been proposed for a wide range of computational problems [16] [5] [26] [3], but they are found to be inadequate for the specific needs of ILP acceleration. One key issue is the use of a 1-bit adder on the column-lines, which proves insufficient for effectively addressing two critical aspects of ILP computation. First, it struggles to accelerate the sparse, control-flow-intense parts of ILP problems, where efficiency is key. Second, it does not provide sufficient throughput, preventing the near-memory logic from fully exploiting its potential. Finally, the existing PIM accelerators do not handle sparsity extremely well. For effective ILP acceleration, it is essential to handle control-flow intensive throughput-heavy sparse operations efficiently.

IV. THE SPARK ARCHITECTURE

A. Learning from shortcomings of prior approaches

Traditional accelerators for linear algebra, sparsity, Ising/Boltzmann models, CGRAs, and ILP-related works struggle with sparsity, control-heavy operations, and data movement between host and accelerator. CPU and GPU optimizations outperform them, with CPUs leading. SPARK reconfigures L1 cache for dot-product compute, adding minimal near-L1 logic across CPU cores for hardware-based sparsity detection, SLE/B&B execution (Fig. 5.a).

B. SPARK's acceleration strategy satisfying 3C criterion

SPARK's acceleration strategy (Fig. 5.b) satisfies the **3C criterion**: (i) **C1**: Optimizes SLE compute throughput via in/near-memory arithmetic operations. (ii) **C2**: Handles sparsity with algorithmic transformation and sparsity-aware compute. (iii) **C3**: Accelerates B&B by optimizing control flow and reusing SLE components. (iv) SPARK is compact, energy-efficient, and easily integrates into SoCs.

C. Choice of tightly integrated over dedicated accelerator

Integrating dedicated accelerators into modern SoCs is challenging due to i) accelerators increasing area overhead/cost, ii) handling control-intensive tasks suited for CPUs, adding overhead to throughput-focused accelerators, iii) high data movement cost between CPU and accelerator, even for small MIPLIB benchmarks, increasing power overhead, and iv) requiring a distinct programming model, unlike adding instructions to an existing ISA, which is adaptable for programmers.

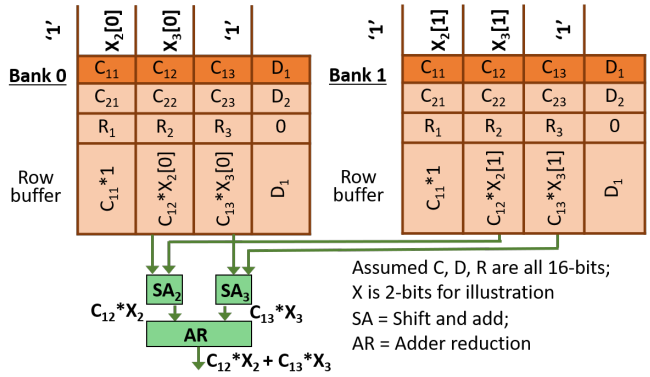


Fig. 7. **In-L1 dot product compute followed by near-L1 accumulation, adder reduction for SLE solving while re-using row buffer/sense amplifier**

D. SPARK's tightly integrated architecture

SPARK reconfigures the L1 cache and the added near memory logic to realize the following engines: i) FC (Fetch/Control) engine for sparsity detection; ii) SA (Sparsity-aware) engine for sparsity-aware compute. iii) SLE engine for SLE acceleration, iv) B&B engine for B&B acceleration.

Micro-architecture overview: Fig. 6 shows SPARK's architecture on a 4-core machine, with n banks of L1 cache per core and near-memory logic shared to realize different engines. The shift-and-add (s-$a_{1\text{-}3}$) operates on partial dot products in memory, with 1 per 16 columns per bank for 16-bit compute, aligning with MIPLIB value ranges. The adder reduction (AR_1) of s-a outputs/subtractors(Sub_1)/dividers(Div_1) are present at a coarse granularity of 1 per bank, with additional counters/queues shared across L1-cache.

E. SPARK's choice of units

SPARK's design focuses on selecting its computational units to minimize area while ensuring that throughput is maintained. In SLE compute, the primary arithmetic operation is MAC, crucial for updating variables by performing MAC across X and C vectors with all other variables (#1 in Fig. 2.a). Because of the importance of the MAC operation in updating these variables, the SA units are designed to operate at a finer granularity to efficiently handle these operations in parallel. In contrast, subtraction and division operations are only needed after all the MAC operations have been computed. These operations require much less complexity and thus can be handled with a single 2-input operation, as illustrated in 2 of Fig. 2b). Therefore, these are present at a coarser granularity. Similar reasoning can be extended to proposed reuse-aware B&B, sparsity-aware ILP compute.

F. SPARK - L1 cache reconfigured for compute

Architecture: The L1 cache in modern processors is typically organized into multiple banks and utilizes 8T SRAM bitcells ([11] [56] [47] [57] [60] [41] [53] [54]), which feature decoupled read/write ports. This configuration allows for efficient data access, as read and write operations can occur simultaneously without interfering with each other. SPARK stores constraint coefficients (C_{ij}, D_i) and cost function (R_i) in L1 cache, reconfiguring it for compute using decoupled read port.

Example:Fig. 7 shows in/near-memory compute for SLE step #1 for X_0 using PIM(L1)+SA+AR. Assuming C, D, R are 16 bits and X is 2 bits, C/D is replicated across 2 banks. The $1^{st}/2^{nd}$ banks handle the dot-product between C (in memory) and the $0^{th}/1^{st}$ bit (mapped to RBL) of X. Row-buffer stores the dot-product result. SA shifts and adds partial products, and AR reduces them to calculate $C_{12}*X_2+C_{13}*X_3$.

G. Circuit details for in-memory compute

L1 cache uses 8T SRAM with decoupled read/write ports (RWL, RBL, WWL, WBL) allowing read-after-write in 2 cycles, compared to 3 cycles for 6T SRAM [45], offering a performance benefit. These have decoupled read (RBL- read bit-line/RWL- read word-line) and write (WBL- write bit line/WWL - write word line) ports, wherein decoupled read-port is "reconfigured" for compute. All columns are computed in 8T SRAM array in parallel.

Dot product compute is performed by storing coefficients onto the bit-cell (SN) and pre-charging RBL based on the value of the incoming variable (X) (Fig. 8.a,b). For X='1'/'0', RBL is precharged to $V_{cc}/(V_{cc}/2)$ respectively. For performing dot product between '1' (SN) and '1' (RBL), RBL discharges below $V_{cc}/2$, while RBL is greater than $V_{cc}/2$ in other cases. RBL discharges via read-port transistors marked in orange, and is sensed by repurposing the sense-amplifier logic to obtain dot product, without modifying array.

H. Choice of L1 cache over last level cache

This choice considered factors like size, performance, throughput, and energy efficiency for MIPLIB benchmarks.

Observation: ∼65% of MIPLIB 2017 benchmarks, fit within a 128KB L1 cache. For 65% of workloads, L1 cache demonstrated: i) superior performance with low access latency, ii) sufficient throughput accommodating the entire workload, and iii) lower energy demands due to reduced data movement compared to constant CPU core requests to LLC. 35% of benchmarks exceeding L1 capacity led to a trade-off analysis between L2 and L1 caches for energy efficiency (throughput divided by energy) in an 8MB LLC versus a 128KB L1 cache.

Results: Simulations show that masking data movement from L2-L1 outweighs LLC's throughput benefits, resulting in 20-25x energy efficiency enhancement in MIPLIB benchmarks. L1 cache was selected, incorporating prefetching to reduce data movement latency while maximizing efficiency.

I. Prefetching for large workloads

Idea: For large workloads that exceed the capacity of the L1 cache, we have implemented a robust prefetching mechanism to effectively address performance and throughput bottlenecks that arise due to cache misses. This strategy takes full advantage of the highly structured and predictable nature of L1 cache accesses. We ensure the accesses to L1 cache start from top to bottom of PIM array, ensuring determinstic compute.

Timeliness/prefetch location: Initiating sequential prefetching requests early enough ensures fill latency amortization in

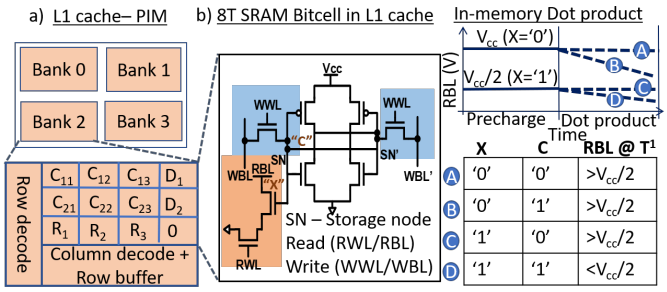


Fig. 8. **a) L1 cache, organized as banks, stores C and cost function (R) consisting of b) 8T SRAM bit cells with decoupled read (orange) and write ports (blue). A data-dependent precharge maps X onto RBL with C stored in bitcell. Dot-product compute between X and C is identified by the value of RBL. RBL at T^1 > Vcc/2 => '0', RBL < Vcc/2 => '1'.**

```
Sparsity aware algorithm
Input:
CC array = Constraints of form Xi <= Di
C array = Constraints not in CC array
Size of C and CC arrays= CN, CCN,
# of constraints(m), variables(n),
Cost Function, C, Const,
D and CC arrays
Cnt initialized to 0
Output:
Solution X and Cost
Sparsity detection:
procedure SPARSE_DETECT(CCN, n)
 if (n==CCN) Sparse=1 //#1
```

```
If deemed sparse; compute the following:
procedure POT_SOLN(CN,CCN,C,D,CC)
 for i=0 to CN do (Initialize k to 0)
    for j=0 to CCN do
      if K!=j then
        Sub[cnt]+= (D[i]-C[i][j]*CC[j])//#1
        PSArr[cnt][j] = CC[j]
    PSArr[cnt][k] = (Sub[cnt]/C[i][k]) //#2
    k++; cnt++;
procedure POT_COSTS(cnt,CCN,PSArr,Cost)
for i=0 to cnt do
   for j=0 to CCN do
     PCArr[i] = PSArr[i][j] * Cost func[j] //#3
Cost = MaxPCArr #4
```

Fig. 9. **Proposed sparsity-aware (SA) algorithm begins with the detection of sparsity in FC engine. If deemed sparse, SA engine executes the proposed SA algorithm, by identifying potential solutions, and costs.**

case of overflow. The choice of a sequential access order is particularly advantageous because the convergence iterations within the computation process are independent of the order in which variables are updated. This independence effectively eliminates the possibility of performance bottlenecks that might otherwise occur if variable update order were a limiting factor. Data is efficiently filled into the L1 cache via the CPU core's dedicated fill pipeline. The data filling process is optimized by replacing the least recently "computed" input constraint, which ensures that the cache is continually populated with the most relevant data for ongoing computations. This approach takes advantage of the L1 cache's ability to support simultaneous read and write operations to different indices, a feature made possible by the decoupled read and write ports in the cache architecture.

J. Impact on traditional CPU workloads

We compared timing metrics with/without near-memory in load-store unit in a 2mm*2mm floorplan and 2ns clock latency in 45nm technology, post place,route, regarding i) increased gate depth affecting critical paths, ii) placement disruptions in conventional logic due to the near-memory logic, iii) latency, access ports, energy, capacity, associativity.

Gate depth: The timing of critical paths in the system remains unaffected by the proposed changes. Specifically, there are three key factors contributing to this: (i) Near-memory logic operates in a separate inactive pipeline during normal operations, and do not add gate depth. The typical critical path for reads involves setting up the read-index (virtual address in VIPT cache), wherein the address can be forwarded from execution units. This path remains unaffected by SPARK. (ii) In the case of processing-in-memory (PIM), no significant modifications are made to the array itself. The only change is the introduction of a 2:1 multiplexer on the periphery, which is used to perform dot-product computations. This multiplexer delay is absorbed with no latency impact (iii) Fill datapath, which gets activated on a fill of a line from higher level caches, is untouched, adding no extra gate depth.

Placement disruptions: There are no additional routing hotspots, congestion leading to placement perturbations, achieved by careful placement of added near memory logic. Most near-memory logic is placed near the fill datapath, which isn't timing critical due to low logic depth, allowing it to tolerate wire delay. Only final stage is near read-datapath, and its low logic depth doesn't disrupt pipelines. Therefore, there's no performance/timing impact from added logic, and SPARK incurs no dynamic power cost as it can be fully gated.

Latency, access ports, energy, capacity, associativity: SRAM latency remains unchanged, as no additional logic is added to the read/write datapath. SPARK's near-memory logic only engages after the computed output from the memory array is captured in the row-buffers (flip-flops).

At the bitcell level, there is no change in access ports. We reuse the decoupled read port in high-performance 8T SRAM for compute, maintaining 1 read, 1 write port. At the array level, the number of compute accesses matches that of simultaneous read accesses, since we reuse the existing row/column-decoding logic, leaving the access ports unchanged.

Read/write energy remains unaffected, as the memory array's read/write datapath isn't altered for compute. The only additional power comes from precharge, due to 2:1 multiplexer for selecting between Vcc/2 and Vcc, adding just 0.001pJ, since it's shared across a column. Associativity, capacity and hit/miss detection circuitry remains unaffected. Like modern processors, we use way-predictor to identify way to access, firing the tag array to confirm correctness. In compute mode, data is "computed" rather than just array read.

Cache Coherence: Among C*,D*,X* stored in memory, cache lines containing C,D do not undergo update, while X undergoes update. Incase lines containing C*/D*/X* get replaced or X* get updated, coherence in traditional CPUs is reused for communicating to other cores. We assume MESI protocol. Memory consistency is unaffected, as existing ordering requirements between memory operations is unaltered.

V. SPARK's FULL-STACK APPROACH

A. Sparsity-aware algorithm

We propose a sparsity-aware algorithm (Fig. 9), explained both mathematically and graphically.

Mathematical understanding: The algorithm starts by detecting sparsity (SPARSE_DETECT) in ILP problems. In an ILP problem with m constraints and n variables ($m \geq n$), the algorithm classifies constraints as either cardinality

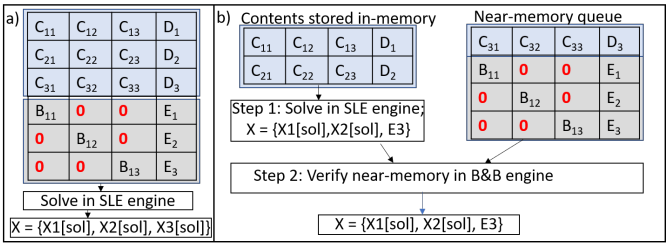


Fig. 10. **Reuse-aware B&B - a) B&B adds sparse constraints (gray) to originally dense (blue) ILPs, and is solved by reusing SLE engine for B&B without dedicated B&B hardware, but suffers from energy-inefficiency. b) Proposed approach overcomes this by having near-memory queues**

constrained (CC) or general. Specifically, constraints of the form $X_i \leq D_i$ are added to the CC array, while other constraints are placed in the general constraints (C) array. When the CC array contains exactly "n" elements, the ILP is considered sparse, indicating that there is a reduced number of active constraints relative to the total possible number. Following sparsity detection, the algorithm proceeds to the POTSOLN function, which identifies potential solutions. This is achieved through efficient dot-product operations performed between the C and CC arrays. After the dot products are computed, subtraction and division steps are applied to further identify potential solutions. Finally, the POTCOSTS function (3) is responsible for determining the maximum and minimum cost values associated with the identified potential solutions.

Graphical understanding: In an n-dimensional space, CC array elements form parallel planes, while C array elements form non-parallel planes. The intersection of these planes gives the optimal solution. Substituting values from n-1 CC array elements into n-1 C array variables yields the n^{th} variable for all constraints (#1, #2 in POT_SOLN). X vectors represent potential solutions, and their costs are potential costs.

B. Reuse-aware B&B algorithm for low area

Idea: To reduce area for B&B acceleration, we propose a reuse-aware approach that allows hardware sharing between B&B and SLE. SPARK reuses the SLE engine for B&B due to the similarities in their computations.

Observation: With each branch, an ILP with n constraints expands to "n+m" constraints, where "m" is the branching tree depth. For an ILP with 3 dense and sparse constraints after 3 B&B levels (n=3), there are two options: i) Reusing the SLE engine for B&B without area overhead, or ii) Adding hardware with a reuse-aware approach.

Tradeoff analysis: The first option (Fig.10a) causes energy inefficiency and SLE under-utilization by solving additional sparse constraints in the SLE engine. The second option (Fig.10b) improves efficiency by solving only the first 2 dense constraints in SLE, while verifying the remaining 4 constraints near-memory, using parallel logic for 3 sparse constraints and MAC for 1 dense constraint. This boosts energy efficiency and compute density in the L1 cache. The near-memory queue for sparse elements further improves energy efficiency by 30% and compute density by 20%.

Instruction	Usage
VFC VS,[Addr]	Detect sparsity in constraints from [Addr], mark sparse VS
VSASLE VC, VX, VS, [Addr]	SA/SLE given VS, constraints from [Addr], VC for cost, writes the result into VX
VBB VB, VX, [Addr]	B&B uses Vx written by VSASLE and constraints from [Addr], writes back to VB, VC

Fig. 11. **SPARK's additional instructions**

Engine	Step/Algorithm	Hardware acceleration method
FC engine	If(n==CCN) Sparse=1 (#1)	**Stage 1**- Fetch constraints ; use near-memory counter to detect sparsity
SA engine	Sub[cnt]+= (D[i] -C[i][j]*CC[j]) (#1)	**Stage 1**- In/near-memory MAC.
	PSArr[cnt][k]=(Sub[cnt]/C[i][k]) (#2)	**Stage 2** – Parallel subtraction, division
	PCArr[i] = PSArr[i][j]*Cost[j](#3)	**Stage 3 –** Near-memory MAC
	Cost = MaxPCArr (#4)	**Stage 4** – Near-memory MAC
SLE engine	sum+= C_{ij}*X_j[n] (#1)	**Stage 1**- Near-memory MAC; **Stage 2**- Adder reduction
	X_j[n+1] = (D_i-sum)/C_{ii} (#2)	**Stage 3** – Parallel subtractors, dividers **Stage 4** – Iter2 queue write of X_j; Parallel execution across j
	L1 norm of X[n+1], X[n] <= Err (#3)	**Stage 5** Copy updated X_j values X_j[n+1] from iter2->iter1 queue + near-memory L1-norm
B&B engine	LB = Calculate_cost(X_n) (#1)	**Stage 1** – PIM/Near-memory MAC
	(var1,var2) = min(frac,LB) (#2a), Dequeue_pruning (queue) (#2b)	**Stage 2a** – Near-memory comparison **Stage 2b –** Parallel nodes invalidation
	Enqueue (X_n, node, parents) ; #3	**Stage 3** – Reuse-aware approach **Stage 4** – Near-memory MAC
	Opt_cost=Cost(node_leaf) #4	**Stage 5 –** Near-memory MAC

Fig. 12. **Acceleration strategy for algorithms in Fig.2, Fig.9**

C. SPARK's programming model

Unlike dedicated accelerators, which often require the use of specialized and complex programming models, SPARK offers a more seamless integration by leveraging existing programming models such as sequential, multithreading, parallel, functional, and others. This flexibility is made possible because SPARK reuses the CPU microarchitecture with minimal additional instructions, meaning it can operate within the frameworks developers are already familiar with. Moreover, the modifications introduced by SPARK occur primarily at the compiler level, meaning that the underlying changes in the system are largely transparent to the programmer. This design choice ensures that the impact on the programming model is minimal, allowing SPARK to work effectively with any existing CPU programming model.

D. SPARK's ISA modifications

SPARK introduces a set of new instructions and registers that significantly enhance the CPU's capabilities in handling specialized tasks. A control register is added that can be programmed to configure L1 cache to compute mode. This can be achieved similar to writing system registers (eg. MSR in ARM). If needed, the system can easily reset this register to return to a non-compute mode, ensuring flexibility and control over the processor's operating state. When in compute mode, new instructions such as VFC, VSASLE, and VBB (Fig.11) are decoded in front-end, signaling the back-end of the core for performing high-throughput near-memory execution [58].

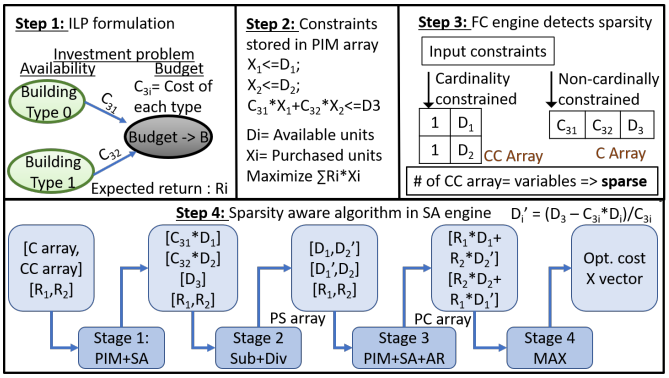


Fig. 13. **Step 1: Investment problem with sparse constraints is stored in L1 cache. Step 2: C matrix and D vector is fetched from the L1 cache in FC engine. Step 3: These are pushed onto either the CC or C array and is used for sparsity detection. Step 4: Sparsity-aware approach uses PIM's high throughput compute between C and CC array.**

E. SPARK's execution strategy

VFC execution: VFC directs the FC engine to detect sparsity, providing ILP sparsity status, crucial for step #1 in Fig. 9. VFC starts by loading constraints into L1 cache. C, D, X vectors can span multiple cache lines without assumptions about their location. Each 64B cache line stores 32 16-bit coefficients. During compute, data is read using base address + offset, and the sparse bit in the VS register is set based on sparsity, repeating across constraints to assess overall sparsity.

VSASLE execution: If deemed sparse, VSASLE is executed in SA engine, else, executed in SLE engine. VX undergoes dot product compute with constraints stored in memory, with VX mapped onto L1 cache columns (Fig.7), for which small (0.001% area overhead) decoding logic is added near-L1 cache. VX stores updated X values post each iteration, mapped onto iteration queues in hardware. During sparsity-aware compute, the address from memory points to either C or CC array elements (#1-2 in Fig.9), and points to constraints for performing Jacobi, in dense compute. SPARK's multiple cache banks help with achieving a high throughput of 32 16-bit MACs possible in a given cycle from a single core for a dense ILP. If a constraint crosses multiple CLs, we perform partial updates for each CL. VC is used for storing the initial cost and stores the updated cost, as we proceed through iterations.

VBB execution: The VBB instruction reads the contents of the VX register and, based on the data, activates the B&B engine for the final ILP solution. This process follows the same principles as those used in the SLE engine, as SLE engine is reused for B&B compute as well. If ILP is sparse, VBB acts as NOP, as B&B engine can be gated during sparse compute.

F. SPARK's execution flow for sparse ILP

FC engine uses near-memory counters to detect sparsity. SA engine uses MAC, subtraction, division to solve sparsity-aware compute. SLE engine uses near-memory MAC, subtraction, and division, B&B engine reuses SLE engine in step 3, near-memory MAC, subtraction, division, discussed in Fig.12.

Step 1 shows ILP formulation for investment problem by using FC and SA engine.

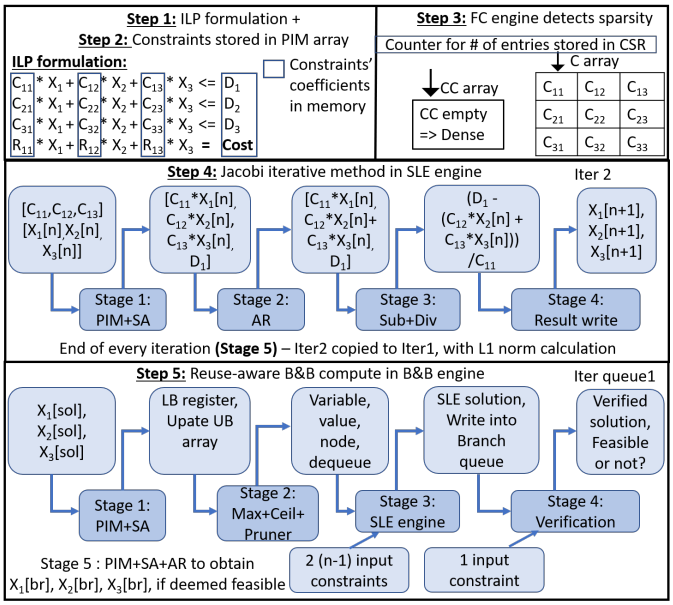


Fig. 14. **ILP with 3 constraints (for example) is stored in the L1 cache in Step 1. In step 2, C and D matrices in the L1 cache are read out and in step 3, the FC engine detects the problem to be dense, as CC array is empty. Jacobi iterative method is executed in Step 4 and the reuse-aware B&B approach in B&B engine accelerates B&B in step 5.**

Step 2 shows PIM (L1 cache) array contents, for illustration Fig. 8.a. If there are n constraints of the form $X_i \leq D_i$ and $(m\text{-}n)$ constraints of the form $\Sigma C_i * X_i \leq D_i$, n constraints of the form are stored first, followed by $m\text{-}n$ constraints, to detect sparsity early and turn off SLE, B&B engines. Fig. 13 illustrates this by storing $X_{1\text{-}2} \leq D_{1\text{-}2}$ followed by $C_{31}*X_1+C_{32}*X_2 \leq D_3$ in Step 2.

Step 3 checks if a constraint is sparse by counting its non-zero coefficients. If there are 2, it goes to the CC array; otherwise, it goes to the C array. The first two constraints store X_i and D_i in the CC array, while the 3rd constraint is pushed to the C array. Sparse constraints are counted in the CC array to determine if the ILP is sparse. The FC engine filters out zero coefficients from C/CC arrays.

Step 4 performs sparsity-aware ILP compute in SA engine using PIM's high throughput. Stage 1 executes a near-memory MAC between elements in C (stored in memory) and CC arrays (mapped to column). Stage 2 finds potential solutions (PS) using near-memory subtractors/dividers. In stage 3, cost of each PS is computed using near-memory MAC and enqueued into the PC array, with optimal cost found in stage 4.

G. Execution flow for dense ILP

Step 1: Fig. 14 shows ILP with 3 dense constraints stored in L1 cache.

Step 2,3: Constraints are read from L1 cache, and the ILP is deemed dense since the CC array is empty, based on the approach mentioned for sparse ILPs.

Step 4: ILP is executed using SLE. Stages 1-2 in SLE engine execute near-memory MAC (Fig. 8c) with stage 3's divider computing step #2 in Fig. 12. Stage 4 updates the result into the Iter2 queue. In stage 5, L1 norm compute

determines whether the problem has converged. The final solution ($X_{1\text{-}3}$[sol]) is transferred to B&B engine.

Step 5: B&B compute for the initial branching tree level is shown using $X_{1\text{-}3}$[sol], assuming the final solution ($X_{1\text{-}3}$[br]) is achieved after one level of branching. In (i) Stage 1, global LB/local UB is calculated by reusing PIM for dot-product and storing in global LB register/local UB array. The local UB array/queue is enqueued after each branching tree level, while global UB remains constant (ii) Stage 2 identifies branching nodes, values, variables, and decisions for parallel pruning using max/ceil/max functions. (iii) Stage 3 capitalizes on parallelism from the SLE engine. (iv) Stage 4 uses reuse-aware approach to verify the solution, by a) reusing PIM, b) using simple MAC without replicating SLE engine. 3 arrays store branching values, variables, and indices of parent nodes, for cases where child node is invalidated along with parent node. (v) In Stage 5, the final solution is obtained through MAC.

H. Execution flow for dense/sparse LPs

Sparse/dense LPs use same flow as their ILP counterparts. In LPs the final solution comes from SLE, as B&B is unused. Revisiting SLE algorithm in Fig. 2, using integers for C/D in-memory enables add/sub/mul operations with the mantissa, given identical exponents across X. Steps #1,#2 are resolved to an integer dot-product/subtraction between mantissa of X and C, with divider in step #2. Steps #1, #2 are repeated till convergence.

VI. EVALUATION METHODOLOGY

A. Benchmarks

MIPLIB 2017/2010 [20] consists of real-life ILPs developed to analyze the performance of different ILP solvers. We chose 7 benchmarks (a mix of L1-cache fitting and overflowing benchmarks) to evaluate SPARK's benefits. For instance, NS, ST, BL do not fit inside the L1 cache, to study SPARK's effectiveness for large-sized workloads.

B. Simulation methodology

SPARK is compared to software-optimized ILP execution on multi-core CPU/GPU, as there are no prior ILP accelerators, and traditional accelerators fall short of CPU/GPU performance (Fig.3). SPARK's performance is modeled using a C++-based cycle-accurate simulator with a prefetching strategy to hide data movement latency, and it simulates PIM array intricacies at the RBL/RWL level, which existing CPU simulators (Gem5) cannot. Python API-based ILP execution provides end-to-end application performance, including setup and data loading effects seen in CPU/GPU execution.

C. SPARK model

SPARK's model includes 32KB I/D cache, 64B cache line size, 4MB shared L2 cache, LRU replacement, 2GB DRAM, 5-wide decode, 8-fetch width, 32-entry load/store queue, and a stride-2 prefetcher for the PIM-capable L1. To validate SPARK's CPU modeling (without PIM mode), we compare to Gem5 using 16 SPEC benchmarks (8 INT/8 FP), covering L1 cache hits (eg. sjeng, hmmer) and misses (eg. bzip2, gcc, gobmk) to evaluate prefetching/memory performance. SPARK's metrics, including L1 cache hits, miss latency, execution time, align within 0.2% of Gem5 results.

D. SPARK micro-architecture

8T SRAM-based L1 cache array is organized into 16 banks, each with 256 rows and 256 columns, optimized for PIM compute. The near-memory logic includes: a) A 32-bit counter in the control stage's cardinality checker for sparsity detection. b) The SLE engine includes two 256-entry centralized arrays repurposed for potential solutions and cost arrays in the SA engine. c) The B&B engine has a shared Global UB/LB register with 1024 entries, including UB/LB and branch variable/value arrays. d) Subtractors/dividers are set at 1 per bank, with adders at 1 per 16 columns. Energy for compute/read is based on RBL discharge when RWL is ON, with 40fF/35fF capacitance at 1V. SRAM latency is 2ns, and data movement costs 1pJ/bit [30]. We describe near-memory logic in System Verilog and synthesize SPARK's digital components for area, power, and energy estimates using Synopsys Design Compiler with 45nm FreePDK technology [51], operating at 1V with a 2ns clock.

E. CPU/GPU comparison

Multi-core CPU is AMD's Zen3, using ILP solver Gurobi, multi-threading/cores and AVX. The GPU used is NVIDIA's Tesla v100, along with cuSparse to solve sparse ILPs. Execution times are recorded for performance comparison. Power for GPU is measured using Nvidia System Management Interface (nvidia-smi). The power in the idle state is deducted from the power associated while executing an ILP to separate power usage from other processes. In CPUs, power is measured using power-stat, discounting idle power. Energy is obtained by multiplying time with power.

F. Performance Breakdown Evaluation

SPARK's benefits come from i) reduced data movement due to in/near-memory compute alongwith prefetching. ii) high throughput of parallel PIM compute. iii) Sparsity-awareness. We identify their relative contributions: For iii), we get rid of sparse datapath and compute using dense datapath. For ii), we model PIM's throughput to be 1 op/cycle to mimic non-parallel PIM compute, while the remaining benefits come from the reduced data movement aspect.

VII. RESULTS

A. Performance comparison for sparse ILP wrt CPU

Fig. 15.a shows the comparison of execution time measured using Spark and software-optimized multi-core CPU (AMD Zen3)/GPU ILP, running MIPLIB benchmarks, considering application-level tradeoffs. The execution times in CPUs are at least 12x-15x higher than in Spark due to reduced data movement, increased throughput in PIM and early detection of sparsity with sparsity-aware execution leading to reduced number of insignificant computations to complete iterations

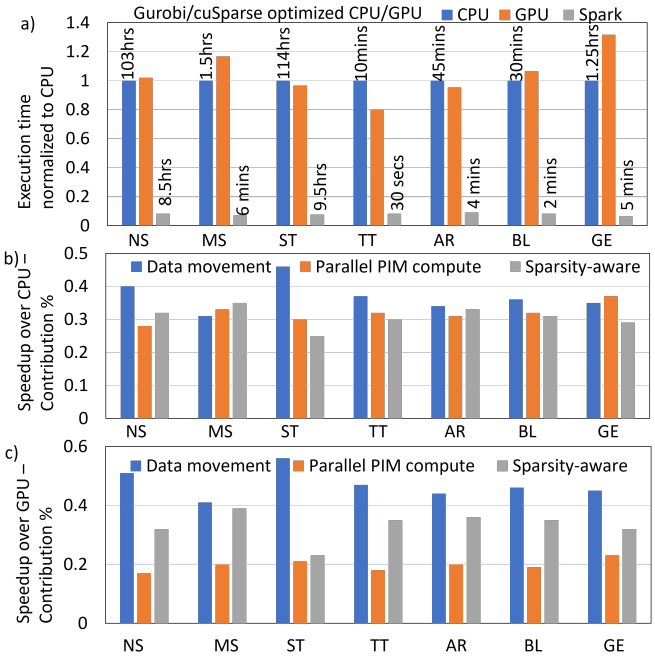


Fig. 15. **Speedup of Spark for sparse ILP: a) Spark shows 12-15x/12-20x speedup over Gurobi/cuSparse optimized CPU/GPU. Relative contribution of reduced data movement, parallel compute, and sparsity-aware compute for improvement over b) CPU c) GPU.**

faster. This is valuable in cases where the sparsity is less (70-80%) like in MS, AR. Fig. 15.b shows that data movement cost in large workloads is higher, parallel PIM compute is useful uniformly across all workloads, sparsity-aware compute is more valuable in highly sparse workloads. Speedup is higher for workloads that fit in L1, due to reduced data movement

B. Performance comparison for sparse ILP wrt GPU

Despite cuSparse optimization, GPU performance lags behind the CPU, while SPARK achieves a 12–20x speedup over the GPU due to (i) the absence of host-GPU interaction overhead, as SPARK integrates seamlessly into the CPU pipeline. (ii) The data transfer overhead for dot product computation is reduced by performing in/near-memory compute. (iii) The near-memory sparsity-aware algorithm minimizes hardware underutilization from sparsity/B&B by performing only useful compute with PIM, as seen in NS, ST, TT, AR, and BL, where sparsity is very high. (iv) cuSparse is ineffective for MS, GE than Gurobi, where sparsity is low, resulting in longer execution time than CPU, achieving 20x speedups. Fig. 15.c shows similar trends as that of Fig. 15.b.

C. Energy comparison for sparse ILP wrt CPU

Spark shows 117-152x energy improvement over CPU, considering power, execution time (Fig. 16). For CPU and Spark, the average power required is approximately 80-90W for the CPU and 7-10W for Spark. This substantial difference in power consumption is primarily due to Spark's design, which significantly reduces data movement, incorporates early sparsity detection, and utilizes a reuse-aware architecture. These optimizations help minimize power consumption in

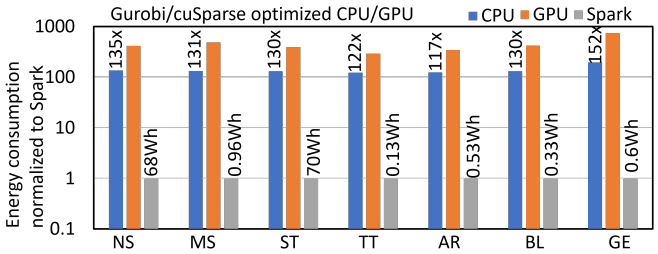


Fig. 16. **Spark shows 117-152x/400-740x improvement in energy for sparse ILP over CPU/GPU. Note: y-axis uses log scale.**

Spark compared to the CPU. In addition to offering significant energy improvements of 120x in extremely sparse workloads, Spark achieves even higher energy savings of 150x in GE, a less sparse workload. This improvement is driven by early sparsity detection, which enables the system to shut off unused engines, further reducing energy usage.

D. Energy comparison for sparse ILP wrt GPU

We observe 400-740x energy improvements over the GPU (which averages 250W). Specifically, the GPU's streaming engine is often underutilized, and there is frequent host-GPU data movement, both of which contribute to inefficiencies in energy usage. Additionally, GPUs lack the specialized execution units required for optimal energy efficiency in certain workloads. These issues are effectively mitigated by leveraging near-memory sparsity and reuse-aware compute strategies, which optimize the computation process directly within memory, reducing unnecessary data movement. For less-sparse workloads like GE, the energy improvements are higher (740x) because of inefficient GPU compute.

E. Performance/energy comparison for sparse LP

We relax integer constraints from MIPLIB benchmarks, removing B&B. GPUs struggle with sparsity and thread divergence. Fig. 17.a shows if CPUs outperform GPUs without Bdivergence, the ILP problem is sparsity-bound (SB). Otherwise, it is divergence-bound (DB). In benchmarks like NS, speedups in solving LPs over CPUs are seen, but not significant, demonstrating the interaction between divergence and sparsity bounds, with divergence dominating sparsity. Spark shows a speedup of 7-20x/8-17x than CPU/GPU due to sparsity aware near-memory compute. For DB benchmarks, the enhanced performance of the GPU makes up for the overall power requirement. For SB benchmarks (MS, GE), CPU power overhead is higher. With SPARK's near-memory sparsity-aware compute, the B&B engine is turned off, yielding 103-272x/96-250x improvements over CPU/GPU (Fig. 17.b).

F. Performance/energy comparison for dense ILP

We run randomly generated dense ILP constraints on CPUs with Gurobi and on GPUs without cuSparse. SPARK's near-memory SLE engine improves efficiency and performance. Fig. 18a shows a 6-8x speedup over the CPU due to limited throughput and high execution latency. Thread divergence in B&B reduces GPU throughput, leading to periodic host-GPU

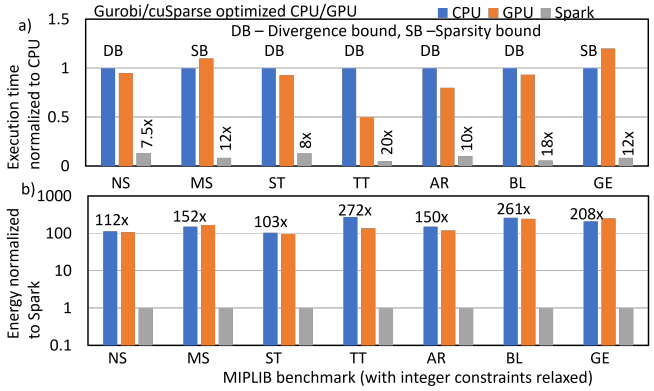


Fig. 17. **Spark vs CPU/GPU for LP - Sparse LP in SA engine- a) Performance b) Energy comparison between Gurobi (CPU) and cuSparse (GPU) decoupling sparsity (both SLE and B&B) and thread divergence issues (B&B) in GPU, as there is no B&B overhead. Spark shows 7-20x/8-17x speedup/energy improvement of 103-272x/96-250x over CPU/GPU.**

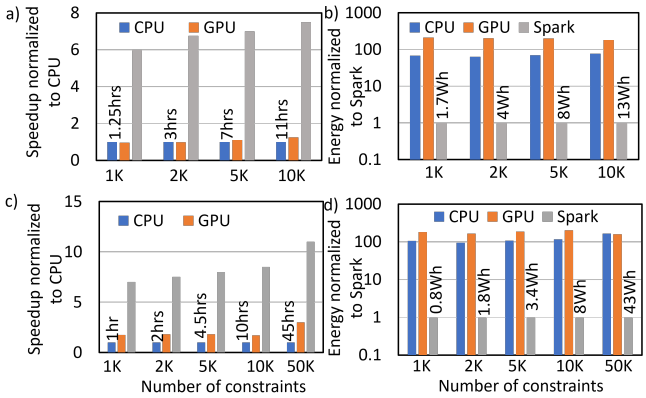


Fig. 18. **a) Dense ILP speedup sensitivity to problem size : Speedup of 6-8x/7-10x b) Energy improvement of 60-75x/180-210x over CPU+Gurobi/GPU c) Dense LP sensitivity: Speedup of 7-7.5x/4-5x over CPU/GPU d) Energy improvement of 105-115x/150-180x over CPU/GPU.**

interactions, causing a 7-10x speedup for SPARK over the GPU. For dense ILP, we observe linear speedup for 1K-10K constraints, as convergence slows with more constraints. Fig. 18b shows a 60-75x/180-210x energy improvement over CPU/GPU with reuse-aware near-memory approach.

G. Performance/energy comparison for dense LP

We relax the integer constraints of dense ILPs, and find the time/power for evaluating SLE, as there is no B&B for LPs. Fig. 18.c shows a 4-5x speedup due to the near-memory high-throughput approach, while GPU requires frequent host-GPU interactions. Jacobi's speedup contribution ranges from 48% to 51% for 1K-50K constraints, with the rest from B&B. The dataset's density increases GPU utilization, leading to better speedups than CPUs. Despite this, GPU power usage increases energy consumption, especially with Gurobi for CPUs up to 10K constraints. Beyond 50K constraints, energy decreases due to better streaming engine utilization. SPARK shows a 105-180x energy improvement over CPU/GPU by shutting off the B&B engine and leveraging near-memory compute.

H. SPARK's area analysis

SPARK's area analysis is divided into three main components: i) memory array (bitcells), ii) peripheral circuitry (row/column decoders, multiplexers, and sensing circuitry), and iii) added near-memory logic (shifters and adders).

Memory array: We use a 16-bank 8T SRAM array with a size of 0.08mm² per core. SPARK doesn't alter the memory array, as the existing RBL and contents are reused for in-memory computation, ensuring that the area occupied by the memory array remains unchanged.

Peripheral circuitry: SPARK introduces a 2:1 multiplexer to enable efficient dot-product computation through RBL precharge. This results in increasing the area by 0.005mm² per core. Other multiplexers/decoders remain unchanged from the baseline peripheral circuitry.

Added near-memory: SPARK requires 0.37mm² per core, including sparsity detection counters (0.03mm²/0.1%), subtractors/dividers (0.11mm²/0.4%), shift-add/adder (0.1mm²/0.4%), comparators (0.02mm²/0.1%), and control/queues (0.11mm²/0.4%). Reduced area is due to resource sharing across SPARK engines.

I. Comparison between Tesla A100 and V100

We simulate using Tesla A100 in addition to the Tesla V100, with the results shown in Fig. 19. While the A100 boasts higher computational power and increased memory bandwidth compared to the V100, the observed performance gains are minimal and only become significant when processing large workloads. For smaller workloads, both GPUs show nearly identical performance in terms of solution time. This is because, despite the A100's higher throughput capabilities, the latency involved in data movement cannot be sufficiently offset by its increased bandwidth in smaller datasets. Furthermore, the A100 consumes more energy due to its higher power requirements, which may reduce its overall energy efficiency for tasks that do not fully leverage its enhanced capabilities, making V100 a power-efficient option for smaller workloads

J. SPARK's performance with varying L1/L2/L3 sizes

Fig.20 shows SPARK's performance variation with L1 size, highlighting the interaction between L1 cache size and cache throughput (determined by read/compute size).

Cache size: For workloads that do not fit in the L1 cache (such as NS, ST, and BL), reducing the size of the L1 cache further leads to a decrease in performance. This is due to the increased number of cache misses and the need to fetch data from slower levels of memory. However, prefetching

MIPLIB Instance	V100 Soln time	A100 Soln time	V100 energy	A100 energy
NS	105 hrs	100 hrs	27.2 KWh	29 KWh
MS	1.75 hrs	1.7 hrs	0.5 KWh	0.7 KWh
ST	110 hrs	103 hrs	28 KWh	30 KWh
TT	8 mins	9 mins	52 Wh	55 Wh
AR	40 mins	45 mins	0.2 KWh	0.4 KWh
BL	35 mins	45 mins	0.1 KWh	0.2 KWh
GE	1.7] hrs	1.6 hrs	0.4 KWh	0.5 KWh

Fig. 19. **A100/V100 comparison**

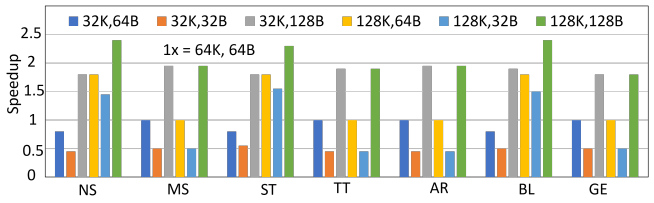


Fig. 20. **Speedup normalized to 64KB L1 cache, read of 64B. Label X,Y implies L1 cache of size X, read of Y**

mitigates most of this performance loss, limiting the overall performance drop to just 0.2x. On the other hand, increasing the L1 cache size for these workloads provides a performance boost, achieving a speedup of up to 1.5x by reducing cache misses and improving memory access times. For workloads that fit entirely within the L1 cache, performance remains unaffected by changes in cache size, as all necessary data is already available in the faster L1 cache.

Read/compute size: L1 cache read/compute size is crucial as it directly affects SPARK's near-memory logic utilization. Halving the throughput typically reduces speedup by half across most workloads. Conversely, doubling throughput, size enables speedups of 2.4x for workloads that don't fit in L1.

For L2 cache, read/compute size is irrelevant since we don't perform near-L2 compute (reserved for follow-up work). Reducing L2 size from 4MB to 2MB halves performance, but prefetching can recover up to 0.9x. L3 cache shows no sensitivity in MIPLIB benchmarks, as they fit within L1, L2.

VIII. Discussion

A. SPARK's importance

Firstly, SPARK accelerates ILP/LP with minimal area overhead by tightly integrating CPUs with L1 cache, exceeding decision thresholds in several benchmarks (TT, AR, BL, GE). It delivers significant speedups over CPU/GPU and lower energy compared to existing methods. Even when the threshold isn't met for some benchmarks, SPARK consumes less energy than CPU/GPU execution, enabling energy-efficient ILP execution. We have developed near-L1, L2 compute, which improves results in benchmarks that doesn't meet threshold, left for future work. Thus, SPARK is a high-performance, energy-efficient architecture and a foundation for other accelerators.

Secondly, there are 2 broadly used architectures for accelerating workloads, chosen based on the workload:
i) Dedicated accelerator - Located farther from the CPU, these are used for specialized workloads, where the cost of moving data is offset by the accelerator's high throughput.
ii) GPU - Depending on the SoC design and programming model, GPU may take precedence of execution over dedicated accelerators but offers similar tradeoffs as that of dedicated accelerators in terms of throughput/data movement.

In SPARK, we demonstrate that workloads like ILP require real-time processing, making near L1-cache compute necessary for energy-efficient, high-performance design. Thus, we propose an architecture tightly integrated with CPU's L1, reconfiguring CPU's L1 for compute.

B. SPARK's generality to different ILP algorithms

SPARK is adaptable to various ILP algorithms, allowing them to be mapped onto it without hardware modifications. In ILP, constraints are typically expressed as $C * X \leq D$, and different algorithms use this to solve problems efficiently.

For a small number of constraints where direct methods like Cramer's rule may be preferred, the task is to solve n-equations with n-variables and check if other equations satisfy the solution. For example, consider 3 constraints of the form $C_{ij} * X_j \leq D_i$. Coefficients are stored in memory, with RBL mapped to C_{ij} to compute the bit-wise dot-product. Using SA and AR, the final dot-product between C_{ii} and C_{ij} is written into queues. Subtraction is then performed by reading from the queues, followed by division to get the final result.

For a larger problem, hypothetically if Gauss-Seidel method [59] is preferred over Jacobi, the lower and upper triangular matrices are stored in memory. X is multiplied by the upper triangular matrix as in Jacobi, and D is subtracted similarly. The divider computes the determinant, and multiplication with the lower triangular matrix follows same approach as Jacobi.

C. Algorithmic insight and regularizing divider for low area

Background: Jacobi's iterative method and B&B seek local optima using the L1 norm but often get stuck in local minima, mitigated by annealing or regularization. Algorithmically, division is the final step in each iteration, followed by regularization, requiring extra hardware. Hardware-wise, dividers are area-intensive and not ideal for near-memory compute.

Idea: We propose using a "regularizing-divider" for regularization by employing less-accurate division through approximate dividers, as in [34] [27]. This replaces costly division with subtracting the first m-bits of mantissa values, with m adjustable based on error. If the error exceeds 1%, a 64B lookup table (shared across memory banks) provides a correction value to refine the subtraction for the final output.

Results: Subtraction approximation reduces gate depth, achieving 0.5ns latency, 0.15pJ energy in FreePDK 45nm technology, enabling energy-efficient single-cycle division. This results in an average error of 0.2% on MIPLIB benchmarks, aiding regularization of updated X values. The lookup table occupies 0.02mm^2 per core, with a subtractor area of 0.04mm^2.

IX. Conclusion

We propose Spark, a near-memory sparsity-aware accelerator that reconfigures L1 cache in CPUs for high-throughput compute, removing redundant computations with a sparsity-aware approach, reuse-aware approach for control-flow intensive operations that helps reduce the near-memory logic area overhead. Spark achieves speedup of 15x/20x and energy benefits of 152x/740x, over AMD's Zen3 CPU/Nvidia's Tesla v100 GPU for real-life sparse MIPLIB 2017 applications. In dense ILPs, Spark achieves 6-10x/60-210x performance/energy improvement over CPU and GPU. Spark achieves 7-17x/103-250x performance/energy improvement over CPU and GPU in sparse LP, and 5-7x/150-180x performance/energy improvement over CPU/GPU in dense LP.

REFERENCES

[1] "[online] introduction to cloud tpu," https://cloud.google.com/tpu/docs/intro-to-tpu.

[2] S. Abdullaev, "The hamilton-jacobi method and hamiltonian maps," *Journal of Physics A: Mathematical and General*, vol. 35, no. 12, p. 2811, 2002.

[3] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 481–492.

[4] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ilp versus specialized 0-1 ilp: An update," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002, pp. 450–457.

[5] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, "Comefa: Compute-in-memory blocks for fpgas," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2022, pp. 1–9.

[6] S. Bayliss, C.-s. Bouganis, G. A. Constantinides, and W. Luk, "An fpga implementation of the simplex algorithm," in *2006 IEEE International Conference on Field Programmable Technology*, 2006, pp. 49–56.

[7] T. Berthold, "A computational study of primal heuristics inside an mi (nl) p solver," *Journal of Global Optimization*, vol. 70, no. 1, pp. 189–206, 2018.

[8] J. Buček, P. Kubalík, R. Lórencz, and T. Zahradnickỳ, "Design of a residue number system based linear system solver in hardware," *Journal of Signal Processing Systems*, vol. 87, pp. 343–356, 2017.

[9] J. BuČek, P. Kubalík, R. Lórencz, and T. Zahradnický, "System on chip design of a linear system solver," in *2014 International Symposium on System-on-Chip (SoC)*, 2014, pp. 1–6.

[10] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1121–1136, 2013.

[11] L. Chang, R. K. Montoye, Y. Nakamura, K. A. Batson, R. J. Eickemeyer, R. H. Dennard, W. Haensch, and D. Jamsek, "An 8t-sram for variability tolerance and low-voltage operation in high-performance caches," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 956–963, 2008.

[12] V. Chvatal, V. Chvatal *et al.*, *Linear programming*. Macmillan, 1983.

[13] I. I. Cplex, "V12. 1: User's manual for cplex," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.

[14] G. B. Dantzig, "Linear programming," *Operations research*, vol. 50, no. 1, pp. 42–47, 2002.

[15] S. Dave and A. Shrivastava, "Ccf: A cgra compilation framework," 2018.

[16] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45Th annual international symposium on computer architecture (ISCA)*. IEEE, 2018, pp. 383–396.

[17] M. Fischetti, A. Lodi, M. Monaci, D. Salvagnin, and A. Tramontani, "Improving branch-and-cut performance by random sampling," *Mathematical Programming Computation*, vol. 8, no. 1, pp. 113–132, 2016.

[18] F. Gao and L. Han, "Implementing the nelder-mead simplex algorithm with adaptive parameters," *Computational Optimization and Applications*, vol. 51, no. 1, pp. 259–277, 2012.

[19] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. Christophel, K. Jarck, T. Koch, J. Linderoth *et al.*, "Miplib 2017: data-driven compilation of the 6th mixed-integer programming library," *Mathematical Programming Computation*, vol. 13, no. 3, pp. 443–490, 2021.

[20] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano, "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library," *Mathematical Programming Computation*, 2021. [Online]. Available: https://doi.org/10.1007/s12532-020-00194-3

[21] G. Gockner, "Guorobi blog," *https://support.gurobi.com/hc/en-us/articles/360012237852-Does-Gurobi-support-GPUs-*, 2023.

[22] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 151–165. [Online]. Available: https://doi.org/10.1145/3352460.3358291

[23] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[24] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[25] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proceedings of the fourth workshop on general purpose processing on graphics processing units*, 2011, pp. 1–8.

[26] B. Hanindhito, R. Li, D. Gourounas, A. Fathi, K. Govil, D. Trenev, A. Gerstlauer, and L. John, "Wave-pim: Accelerating wave simulation using processing-in-memory," in *Proceedings of the 50th International Conference on Parallel Processing*, 2021, pp. 1–11.

[27] S. Hashemi, R. I. Bahar, and S. Reda, "A low-power dynamic divider for approximate applications," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[28] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, "Sparse-tpu: Adapting systolic arrays for sparse matrices," in *Proceedings of the 34th ACM international conference on supercomputing*, 2020, pp. 1–12.

[29] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[30] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14.

[31] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 554–566.

[32] Q. Huangfu and J. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, 2018.

[33] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 968–981.

[34] M. Imani, R. Garcia, A. Huang, and T. Rosing, "Cade: Configurable approximate divider for energy efficiency," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 586–589.

[35] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589350

[36] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[37] V. Klee and G. J. Minty, "How good is the simplex algorithm," *Inequalities*, vol. 3, no. 3, pp. 159–175, 1972.

[38] T. Koch, T. Berthold, J. Pedersen, and C. Vanaret, "Progress in mathematical programming solvers from 2001 to 2020," *EURO Journal on Computational Optimization*, p. 100031, 2022.

[39] T. Koch, A. Martin, and M. E. Pfetsch, "Progress in academic computational integer programming," in *Facets of Combinatorial Optimization*. Springer, 2013, pp. 483–506.

[40] T. Koch, T. Ralphs, and Y. Shinano, "Could we use a million cores to solve an integer program?" *Mathematical Methods of Operations Research*, vol. 76, no. 1, pp. 67–93, 2012.

[41] J. Kulkarni, M. Khellah, J. Tschanz, B. Geuskens, R. Jain, S. Kim, and V. De, "Dual-v cc 8t-bitcell sram array in 22nm tri-gate cmos for energy-efficient operation across wide dynamic voltage range," in *2013 Symposium on VLSI Technology*. IEEE, 2013, pp. C126–C127.

[42] L. Liberti, "Symmetry in mathematical programming," in *Mixed Integer Nonlinear Programming*. Springer, 2012, pp. 263–283.

[43] A. Lucas, "Ising formulations of many np problems," *Frontiers in physics*, vol. 2, p. 5, 2014.

[44] F. Margot, "Exploiting orbits in symmetric ilp," *Mathematical Programming*, vol. 98, no. 1, pp. 3–21, 2003.

[45] S. S. T. Nibhanupudi, S. R. S. Raman, and J. P. Kulkarni, "Phase transition material-assisted low-power sram design," *IEEE Transactions on Electron Devices*, vol. 68, no. 5, pp. 2281–2288, 2021.

[46] S. Pal, S. Feng, D.-h. Park, S. Kim, A. Amarnath, C.-S. Yang, X. He, J. Beaumont, K. May, Y. Xiong *et al.*, "Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 175–190.

[47] S. R. S. Raman, S. Xie, and J. P.Kulkarni, "Compute-in-edram with backend integrated indium gallium zinc oxide transistors," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[48] E. Reggiani, C. R. Lazo, R. F. Bagué, A. Cristal, M. Olivieri, and O. S. Unsal, "Bison-e: A lightweight and high-performance accelerator for narrow integer linear algebra computing on the edge," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 56–69.

[49] G. L. Sleijpen and H. A. Van der Vorst, "A jacobi–davidson iteration method for linear eigenvalue problems," *SIAM review*, vol. 42, no. 2, pp. 267–293, 2000.

[50] P. Somol, P. Pudil, and J. Kittler, "Fast branch bound algorithms for optimal feature selection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 7, pp. 900–912, 2004.

[51] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh *et al.*, "Freepdk: An open-source variation-aware design kit," in *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*. IEEE, 2007, pp. 173–174.

[52] Y. Su, H. Kim, and B. Kim, "31.2 cim-spin: A 0.5-to-1.2v scalable annealing processor using digital compute-in-memory spin operators and register-based spins for combinatorial optimization problems," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 480–482.

[53] S. R. Sundara Raman, L. John, and J. P. Kulkarni, "Nem-gnn: Dac/adc-less, scalable, reconfigurable, graph and sparsity-aware near-memory accelerator for graph neural networks," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, May 2024. [Online]. Available: https://doi.org/10.1145/3652607

[54] S. R. Sundara Raman, L. John, and J. P. Kulkarni, "Nem-gnn: Dac/adc-less, scalable, reconfigurable, graph and sparsity-aware near-memory accelerator for graph neural networks," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, May 2024. [Online]. Available: https://doi.org/10.1145/3652607

[55] S. R. Sundara Raman, L. K. John, and J. P. Kulkarni, "Sachi: A stationarity-aware, all-digital, near-memory, ising architecture," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 719–731.

[56] S. R. Sundara Raman, S. S. T. Nibhanupudi, and J. P. Kulkarni, "Enabling in-memory computations in non-volatile sram designs," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 12, no. 2, pp. 557–568, 2022.

[57] S. R. Sundara Raman, S. Xie, and J. P. Kulkarni, "Igzo cim: Enabling in-memory computations using multilevel capacitorless indium–gallium–zinc–oxide-based embedded dram technology," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 8, no. 1, pp. 35–43, 2022.

[58] J. Turley, "Tensilica cpu bends to designers' will," *Microprocessor Report*, vol. 13, no. 3, p. 12, 1999.

[59] M. Usui, H. Niki, and T. Kohno, "Adaptive gauss-seidel method for linear systems," *International Journal of Computer Mathematics*, vol. 51, no. 1-2, pp. 119–125, 1994.

[60] S. Vangal, S. Paul, S. Hsu, A. Agarwal, S. Kumar, R. Krishnamurthy, H. Krishnamurthy, J. Tschanz, V. De, and C. H. Kim, "Wide-range many-core soc design in scaled cmos: Challenges and opportunities," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 5, pp. 843–856, 2021.

[61] L. A. Wolsey, "Heuristic analysis, linear programming and branch and bound," in *Combinatorial Optimization II*. Springer, 1980, pp. 121–134.

[62] S. Xie, S. R. S. Raman, C. Ni, M. Wang, M. Yang, and J. P. Kulkarni, "Ising-cim: A reconfigurable and scalable compute within memory analog ising accelerator for solving combinatorial optimization problems," *IEEE Journal of Solid-State Circuits*, pp. 1–13, 2022.

[63] R. Xu, S. Ma, Y. Guo, and D. Li, "A survey of design and optimization for systolic array based dnn accelerators," *ACM Computing Surveys*, 2023.

[64] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, "Sara: Scaling a reconfigurable dataflow accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1041–1054.