RSQP: Problem-specific Architectural Customization for Accelerated Convex Quadratic Optimization

Maolin Wang
maolinwang@ust.hk
AI Chip Center for Emerging Smart Systems
Hong Kong

Ian McInerney
i.mcinerney17@imperial.ac.uk
The University of Manchester
United Kingdom

Bartolomeo Stellato
bstellato@princeton.edu
Princeton University
United States

Stephen Boyd
boyd@stanford.edu
Stanford University
United States

Hayden Kwok-Hay So
hso@eee.hku.hk
University of Hong Kong
Hong Kong

ABSTRACT

Convex optimization is at the heart of many performance-critical applications across a wide range of domains. Although many high-performance hardware accelerators have been developed for specific optimization problems in the past, designing such accelerator is a challenging task and the resulting computing architecture is often so specific to the targeted application that they can hardly be reused even in a related application within the same domain. To accelerate general-purpose optimization solvers that must operate on diverse user input during run time, an ideal hardware solver should be able to adapt to the provided optimization problem dynamically while achieving high performance and power-efficiency. In this work, a hardware-accelerated general-purpose quadratic program solver, called RSQP, with reconfigurable functional units and data path that facilitate problem-specific customization is presented. RSQP uses a string-based encoding to describe the problem structure with fine granularity. Based on this encoding, functional units and datapath customized to the sparsity pattern of the problem are created by solving a dictionary-based lossless string compression problem and a mixed integer linear program respectively. RSQP has been integrated to accelerate the general-purpose quadratic programming solver OSQP and has been tested using an extensive benchmark with 120 optimization problems from 6 application domains. Through architectural customization, RSQP achieves up to 7× performance improvement over its baseline generic design. Furthermore, when compared with a CPU and a GPU-accelerated implementation, RSQP achieves up to 31.2× and 6.9× end-to-end speedup on these benchmark programs, respectively. Finally, the FPGA accelerator operates at up to 6.6× lower dynamic power consumption and up to 22.7× higher power efficiency over the GPU implementation, making it an attractive solution for power-conscious datacenter applications.

CCS CONCEPTS

• **Mathematics of computing → Quadratic programming**; • **Hardware → Hardware accelerators**.

KEYWORDS

convex optimization, FPGA, quadratic programming, domain-specific architectures, reconfigurable computing

ACM Reference Format:
Maolin Wang, Ian McInerney, Bartolomeo Stellato, Stephen Boyd, and Hayden Kwok-Hay So. 2023. RSQP: Problem-specific Architectural Customization for Accelerated Convex Quadratic Optimization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579371.3589108

1 INTRODUCTION

Convex optimization is at the heart of many important problems across a wide range of application domains. Specifically, convex quadratic programs (QPs) can be found in many real-world application domains including control engineering [1, 12, 25], financial trading/investment planning [5, 6, 23], data assimilation (e.g. least-squares, lasso and ridge regression) [7] and the optimization subproblems solved when using the Sequential Quadratic Programming (SQP) method for solving more general nonlinear or non-convex optimization problems [16]. QPs can also be integrated as individual layers within deep neural networks. [3] and [2] have demonstrated that QP-based layers are capable of capturing more complex data dependencies compared to traditional convolutional and fully-connected layers.

A convex quadratic program with n decision variables and m constraints can be written as

$$\begin{aligned} &\text{minimize} \quad && (1/2)x^T P x + q^T x && \text{(1a)} \\ &\text{subject to} \quad && l \le Ax \le u, && \text{(1b)} \end{aligned}$$

where $x \in \mathbb{R}^n$ is the vector of decision variables, the positive semi-definite matrix $P \in \mathbb{S}^n_+$ and vector $q \in \mathbb{R}^n$ define the objective, and the matrix $A \in \mathbb{R}^{m \times n}$ and vectors $l, u \in \mathbb{R}^m$ describe the constraints. The QP problem is prevalent in many domains because it can handle a wide range of optimization problems with quadratic objective functions and linear constraints. Quadratic objective functions are popular because they can accurately capture second-order effects and provide a good approximation to non-linear functions. Linear constraints, on the other hand, are easy to work with and widely applicable in many domains. While all QP problems can be expressed in the form of (1), each individual optimization *problem* can differ in the *structure* of the matrices P and A, defined here as the locations of the non-zero values in the matrices. This means that if the QP (1) is solved using an algorithm that uses sparse matrix operations, the requirements to efficiently accelerate the core operations such as sparse matrix-vector multiplication (SpMV) will be different for every problem. For this reason, applications that demand the highest performance in solving their QP routines have typically relied on tailor-made hardware accelerators that are optimized for a particular application [8, 14, 17, 19, 20, 26]. Unfortunately, the design of these hardware accelerators usually requires advanced knowledge of the problem domain, and the resulting hardware designs are not readily transferable between different problem types. Instead, the ideal general-purpose optimization solver should be able to automatically produce a high-quality accelerator architecture by analyzing the user-provided problem without requiring major user interaction and domain expertise.

In this work, we present RSQP, an FPGA-based reconfigurable quadratic programming solver that can customize its architecture to the target problem automatically for improved acceleration. RSQP provides end-to-end acceleration of the Preconditioned Conjugate Gradient (PCG) methods within the general-purpose QP solver OSQP [30]. Given a QP in the form of (1), RSQP uses a string-based encoding to represent the sparsity structure of the matrices and vectors. This encoding is then used to generate customized functional units and datapaths for the user's optimization problem by solving a dictionary-based lossless string compression problem and a mixed integer linear programming problem, respectively.

RSQP was implemented on FPGA platforms designed for data center applications. Extensive experiments were conducted using 120 problems across 6 applications with dimensions ranging from less than 10^2 to over 10^6 number of non-zeros (nnz) automatically generated from the OSQP benchmark set. With architectural customization, RSQP achieves 1.4 to 7.0 times improvement in end-to-end execution speed over the baseline reference design. When compared to the original CPU implementation with MKL acceleration and a GPU-accelerated implementation of OSQP [29], RSQP achieved up to 31.2 and 6.9 times end-to-end speedup, respectively. Moreover, while the GPU consumed 44 W to 126 W across the benchmark, the FPGA accelerator operated at around 19 W in all cases, representing up to 22.7× improvement in power-efficiency across a wide range of problems.

Although customizing the accelerator architecture to a particular QP problem incurs a lengthy hardware implementation process due to the vendor CAD tools, such overhead can be amortized when the resulting accelerator is reused to solve different *instances* of the same optimization problem with the same sparsity structure. For example, up to 120 000 QP problems with the same sparsity structure would need to be solved with different sets of trading strategy dependent parameters to perform backtesting of portfolio optimization over 2 years of historical data [28]. This translates to over 300 hours of run time even if it took only 1 s to solve each QP problem, amortizing the 2 to 5 hours of hardware CAD tools overhead.

RSQP is integrated as part of the OSQP solver, and has been tested with CVXPY [11]. It can readily be deployed to accelerate real-world quadratic programming problems either by using a baseline architecture running on the FPGA or by generating a problem-specific architecture that provides additional performance and power-efficiency improvements. As such, we consider the main contributions of our work to be the following:

- we present a first-of-its-kind general-purpose FPGA-accelerated QP solver that is suitable for deployment in datacenters to accelerate real-world problems;
- we propose an architectural customization framework to adapt the accelerator's datapath and data packing scheme to the structure of the user input that results in substantial performance and power-efficiency improvements; and
- we demonstrate through extensive experiments the superior performance and power-efficiency of the proposed design over existing CPU and GPU accelerated implementations across a wide range of problems.

The rest of the paper is organized as follows. In Section 2 we present the necessary background on the OSQP algorithm and the preconditioned conjugate gradient method. We then present the overall architecture of RSQP and the proposed architectural customization methods in Sections 3 and 4, respectively. Experimental results are shown in Section 5, followed by a discussion of the related work in Section 6. Finally, we conclude our work in Section 7.

2 BACKGROUND

This work builds on the OSQP solver [30], which is a first-order method for solving convex quadratic programs based on the Alternating Direction Method of Multipliers (ADMM) technique. In this section, we present a description of the algorithm implemented in the OSQP solver, and the linear system solvers that are used inside OSQP and RSQP.

2.1 The OSQP algorithm

In OSQP, problem (1) is solved by introducing a new decision variable $z \in \mathbb{R}^m$ that converts the inequality constraint (1b) into the pair of equality/inequality constraints $Ax = z$ and $l \leq z \leq u$ instead. Then, two new sets of auxiliary variables, $\tilde{x} \in \mathbb{R}^n$ and $\tilde{z} \in \mathbb{R}^m$, are introduced. The method used by OSQP for solving (1) is shown in Algorithm 1, and performs two main computations during each iteration: (i) the solution of an equality constrained QP by solving the linear system (2), and (ii) the Euclidean projection, $\Pi(\cdot)$, onto the inequality constraint set.

The first step in each iteration of OSQP is the solution of the equality constrained QP

$$\begin{aligned} \text{minimize} \quad & (1/2)\tilde{x}^T P\tilde{x} + q^T\tilde{x} + (\sigma/2)\|\tilde{x} - x^k\|_2^2 \\ & + (\rho/2)\|\tilde{z} - z^k + \rho^{-1}y^k\|_2^2 \\ \text{subject to} \quad & A\tilde{x} = \tilde{z}, \end{aligned}$$

where $y, v \in \mathbb{R}^m$ are the Lagrange multipliers associated with the constraints, $\sigma > 0$, $\rho > 0$ are the step-size parameters, and

Algorithm 1 OSQP algorithm

```
1: given:
       initial values $x^0$, $z^0$, $y^0$ and parameters $\rho > 0$,
       $\sigma > 0$, $\alpha \in (0, 2)$
2: repeat
3:     solve
           $\begin{bmatrix} P + \sigma I & A^T \\ A & -\rho^{-1} I \end{bmatrix} \begin{bmatrix} \tilde{x}^{k+1} \\ \nu^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \rho^{-1} y^k \end{bmatrix}$   (2)
4:     $\tilde{z}^{k+1} \leftarrow z^k + \rho^{-1}(\nu^{k+1} - y^k)$
5:     $x^{k+1} \leftarrow \alpha \tilde{x}^{k+1} + (1 - \alpha) x^k$
6:     $z^{k+1} \leftarrow \Pi(\alpha \tilde{z}^{k+1} + (1 - \alpha) z^k + \rho^{-1} y^k)$
7:     $y^{k+1} \leftarrow y^k + \rho(\alpha \tilde{z}^{k+1} + (1 - \alpha) z^k - z^{k+1})$
8: until the termination criterion is satisfied
```

$\alpha \in (0, 2)$ is the relaxation parameter (by default, OSQP sets $\alpha = 1.6$ and $\sigma = 10^{-6}$). This QP can be efficiently solved by forming the Karush-Kuhn-Tucker (KKT) matrix of the optimality conditions, and then applying a linear system solver to the KKT system (2). Next, the additional decision variables z are projected into the inequality constraints using the Euclidean projection operation, which for problem (1) is simply the element-wise projection into $[l, u]$, giving $\Pi(z) := \min(\max(z, l), u)$.

2.2 Solving the KKT system

The core component of the OSQP solver is a linear system solver to find the solution of the KKT system (2) at each iteration, with the remaining steps using only straightforward vector operations. Solving the KKT system (2) requires the majority of the time in an OSQP iteration, and can comprise over 95 % of the total computation time (Figure 8).

The KKT system (2) is an indefinite linear system, and OSQP solves it using a direct LDL^T factorization method designed for sparse linear systems. In this method, the KKT matrix is first factorized into a lower triangular matrix L and a diagonal matrix D, and then the solution is found using a forward-backward substitution. OSQP implements this as a three-stage process, with an initial symbolic factorization to determine the sparsity pattern and structure of the L matrix performed before the iterations begin, and then each iteration performing a numerical factorization to actually compute L and D, followed by the forward-backward substitution to compute the final solution. If the values of σ and ρ don't change between iterations, the numerical factorization can be reused in the next iteration, reducing the solution of (2) to only the forward-backward substitution step. Updating ρ between iterations can improve the rate of convergence, so by default OSQP will occasionally update ρ, necessitating the recomputation of the numerical factorization in the following iteration.

An alternative way to solve the KKT system (2) is to reduce it into the symmetric linear system

$$(P + \sigma I + \rho A^T A)\tilde{x}^{k+1} = \sigma x^k - q + A^T(\rho z^k - y^k), \qquad (3)$$

and then solve for $\tilde{x}^{k+1}$. For convenience, we let $K := (P + \sigma I + \rho A^T A)$ be the matrix from (3), and refer to K as the reduced KKT matrix.

Algorithm 2 Preconditioned Conjugate Gradient

```
1: given:
       initial values $x^0$ and preconditioner $M$
       $K := P + \sigma I + \rho A^T A$
       $b := \sigma x^k - q + A^T(\rho z^k - y^k)$
       $r^0 := K x^0 - b$,   $d^0 := M^{-1} r^0$,   $p^0 := -d^0$
2: repeat
3:     $\lambda^{i+1} \leftarrow \frac{(r^i)^T d^i}{(p^i)^T K p^i}$
4:     $x^{i+1} \leftarrow x^i + \lambda^i p^i$
5:     $r^{i+1} \leftarrow r^i + \lambda^i K p^i$
6:     $d^{i+1} \leftarrow M^{-1} r^{i+1}$
7:     $\mu^{i+1} \leftarrow \frac{(r^{i+1})^T d^{i+1}}{(r^i)^T d^i}$
8:     $p^{i+1} \leftarrow -d^{i+1} + \mu^{i+1} p^i$
9:     $i \leftarrow i + 1$
10: until $||r^i|| < \epsilon ||b||$
```

The reduced KKT matrix K in OSQP will always be positive-definite, allowing the reduced system (3) to be solved using the Preconditioned Conjugate Gradient (PCG) iterative method, which was done in cuOSQP [29], the CUDA-based GPU implementation of OSQP. An iteration of PCG is shown in Algorithm 2, and requires only a single matrix-vector product with K and other vector operations, making the PCG simpler and easier to implement compared to the LDL^T factorization solver. When using PCG to solve (3), K should never be explicitly computed, since forming the matrix-matrix product $A^T A$ could destroy any structure/sparsity that the problem originally possessed and require storing a dense matrix for K. Therefore, it is better to store P, A, and A^T separately and then perform the matrix-vector multiplication Kp^i in an incremental manner.

The use of PCG in OSQP has several advantages over the LDL^T direct factorization method. Firstly, using PCG allows for more frequent updates of ρ, since K is never formed explicitly and the numerical refactorization step from the LDL^T solver is avoided. Secondly, the computation cost of PCG is much lower than the computation cost of the LDL^T factorization steps in an OSQP iteration for large and sparse KKT matrices. Finally, PCG can present more opportunities for parallelization and performance tuning, making it better for embarrassingly parallel architectures like GPUs and FPGAs.

While the GPU implementation of OSQP utilized PCG [29], it was only able to take partial advantage of these benefits, since it is difficult to fully exploit problem-specific structures on the GPU's homogeneous architecture. In this work, we also adopt the PCG method to solve the reduced KKT system (3), but additionally introduce a methodology for designing problem-specific acceleration architectures to speed up the computation of the matrix-vector product.

3 THE RSQP FRAMEWORK

RSQP encompasses both the processing architecture running on the FPGA accelerator, as well as the software framework that performs problem-specific architectural customization and integrates with the rest of the software solver. In this section, an overview of the entire framework is first provided, while the detailed customization algorithms are discussed in the next section.

3.1 Processing Architecture Overview

Figure 1 shows an overview of RSQP's reconfigurable processing architecture that runs on the FPGA. In this architecture, the high bandwidth memory (HBM) of the FPGA serves as the main data exchange between the CPU and the FPGA. The problem data matrices P, A and A^T, the right hand side of (3), as well as results from the accelerator are all passed between the CPU and the FPGA through HBM.

The high bandwidth of HBM is essential to achieve high performance on the accelerator since the performance of the core sparse matrix-vector multiply (SpMV) and vector-vector operations is limited by the memory bandwidth. The datapath of RSQP consists of a configurable sparse matrix-vector multiply (SpMV) engine, a vector engine, an on-chip vector buffer (VB), as well as a special compressed vector buffer (CVB). The widths of the data paths connecting these major components are uniformly determined by a configurable data width parameter C. The vectors in the algorithms are partitioned across C VBs for parallel access in sequence. As a result, the number of clock cycles needed to complete the vector operations and data transferring instructions is inversely proportional to C. Vectors related to the decision variables $(x, \tilde{x}, b, r, d, u, p)$ have length n, while vectors related to constraints (z, v, y) have length m. The clock cycles required for their dot product, element-wise operations and data movement are thus n/C and m/C respectively. As such, the data width parameter C can be effectively used to tune the level of parallelism for problems of different scales. To solve large problems faster, we can instantiate RSQP with a larger C.

The sparse matrices P, A and A^T of the problem, represented by the non-zero values and their coordinates, are partitioned across different HBM channels for high throughput parallel access. Maintaining a high computation throughput of the SpMV engine is crucial for the speed of the OSQP algorithm. In order to provide the desired performance for a wide range of optimization problems, the data path and functional units of RSQP's SpMV engine are designed to be flexible and customizable. In particular, the following architectural features, highlighted by colors in Figure 1, are all designed to be parametrizable and will be optimized for each user-provided problem:

- the multiplier-adder inter-connection within the SpMV functional unit,
- the routing logic between the SpMV engine and VBs, and
- the compressed Vector Buffers (CVB), which provide parallel random access to the vector of the multiplication.

3.2 Customizable Data Paths within the SpMV Engine

The SpMV operation is a memory-bounded operation since every non-zero value of the matrix stored on the HBM is used only once per SpMV operation. In order to read C non-zero values of the problem matrices from the HBM at every clock cycle, the access pattern needs to be contiguous. In the simplest case, all non-zero value packs of length C fed from the HBM at each clock cycle belong to the same matrix row. In this case, the SpMV Engine simply needs to perform a C-input-single-output Multiply-and-Accumulate (MAC) to produce a complete dot product between the matrix row and the vector every clock cycle. This ideal baseline multiplier-adder connection within the SpMV engine can benefit from using a balanced binary tree structure to provide the desired throughput.

Unfortunately, in real-world problems, data fragmentation due to the use of large values for C and the great degree of variation in matrix sparsity patterns can rapidly reduce the utilization efficiency of the SpMV engine. The non-zero values in the pack provided to the SpMV Engine at each clock cycle are very likely to belong to different matrix rows. Consequently, since additional zeros are needed to be inserted to break the pack into different rows, the ideal baseline MAC unit with a balanced reduction tree structure will not be able to complete the dot products of all matrix rows contained in the non-zero pack with the vector in each clock cycle.

If we have some prior knowledge about the row distributions contained in the non-zero packs, connections of the multipliers and adders can be customized to deal with the known distribution efficiently. For example, if there are many non-zero packs containing just two matrix rows with lengths no greater than $C/2$, we can add dedicated output data paths to the intermediate adders. Then the non-zero pack can be sent to the MAC tree for computing two dot products directly without inserting extra zeros. If non-zero packs containing 4 rows are common, we can add another set of dedicated output data paths to adders in even earlier stages. The detailed methodology of extracting row distribution patterns and designing customized MAC connections automatically will be discussed in Section 4.

3.3 Aligning the SpMV Results

The customized data paths inside the SpMV engine lead to variable result output lengths. RSQP has a routing module between the SpMV Engine and the VB to align the variable result lengths back to width C for subsequent storage and vector operations. Given the customization inside the SpMV engine, the companion routing logic can be generated automatically.

3.4 Compressed Vector Buffers

The CVB in Figure 1 provides parallel random access to C locations of the vector to be multiplied with the matrix elements in every clock cycle. Providing the ability of multiple random accesses through simple duplication of the vector by C copies has severe scalability pressure on the capacity of the on-chip memory. Besides, the vector needs to be updated by other steps in Algorithm 1 and Algorithm 2 after every MV operation. The time spent on vector duplication instructions will increase significantly if we have too many copies of the vector in the CVB, so the CVB requires a customized design to provide the desired parallel access capability with a low profile.

3.5 Control

The operation of the RSQP processing architecture is controlled by a simple instruction unit. Table 1 lists the instructions supported by RSQP and their usage when implementing Algorithm 1 and Algorithm 2 on this architecture. The algorithms are broken into a

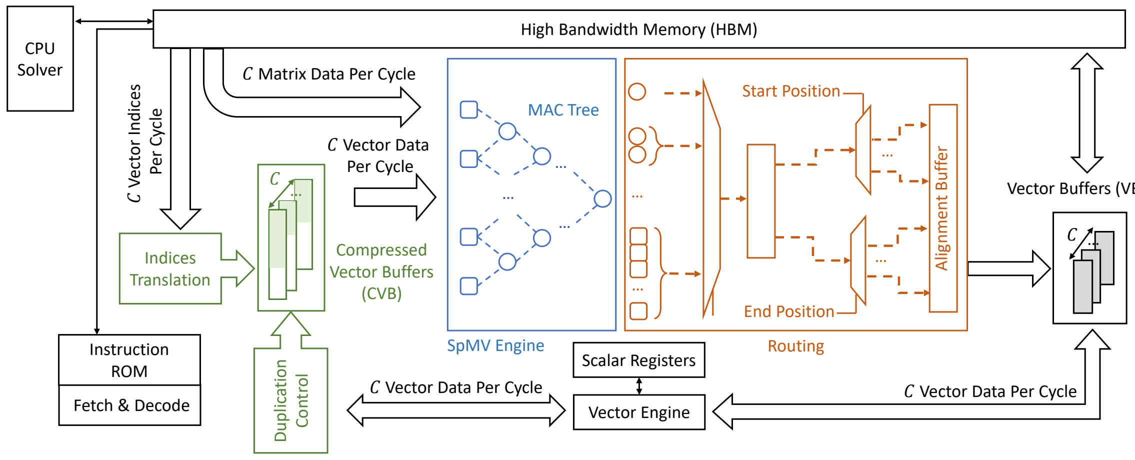

Figure 1: Architecture Design

Table 1: Instruction Set

Instruction Type	Function	Usage
Control	Exit the algorithm loop if residual is less than threshold	A1-8, A2-10
Scalar Arithmetic	Addition, subtraction, division, multiplication	A2-3,7,9
Data transfer	Read/write a vector from/to memory	A2-1,10
Vector Operations	Linear combination of two vectors, element-wise comparison/reciprocal/multiplication, dot product	A1-4,5,6,7,A2-1,3,4,5,6,7,8
Vector Duplication	Duplicate vector copies across buffers	A2-1,3
SpMV	Multiply a matrix with a vector and write the result to the vector buffer	A1-8, A2-1,3

sequence of instructions which will be downloaded to the instruction ROM from HBM. The instructions activate the vector engine, SpMV engine, and data movement modules of RSQP as needed. Each vector operation instruction takes a complete vector from a CVB and another vector from a VB. The scalar result of the vector operation instruction is written to scalar registers. If the result is a vector, it can be written to a CVB or a VB based on the field of the instruction. The SpMV instruction takes the matrix input and vector indices from HBM and vector input from a CVB. The vector indices are used to find the correct location of the vector in the CVB. The SpMV result is written to the VB. Each instruction can only start after the previous instruction has completed.

3.6 Customization Evaluation Metric

Before diving into the methodology of customizing the MAC tree and CVBs, we first describe a metric that RSQP uses to evaluate the effect of customization on the performance. Imagine we have an ideal architecture that has the perfect customization for a specific problem sparsity structure. It only needs one copy of the vector to provide the desired parallel random access capability and can finish the dot products of all the matrix rows contained in the non-zero values pack fed at every clock cycle regardless of their distribution across different matrix rows. Denote the number of non-zeros in the matrix as nnz and the length of the vector as L, then a single run of the SpMV and vector duplication instruction on the ideal architecture will take $T_{\text{img}} = \frac{\text{nnz}+L}{C}$ clock cycles.

Now consider an architecture customization that may not match the problem sparsity structure very well. For the SpMV instruction, it needs to insert extra zeros to the non-zero values and indices streamed in from HBM at every clock cycle to handle various matrix row distributions. Denote the total number of extra zero padding as E_p. For the vector duplication operation, it may store E_c extra copies of the vector in the CVB for parallel access. A single run of the SpMV and vector duplication instruction on this architecture variation will take $T_{\text{real}} = \frac{\text{nnz}+E_p+E_cL}{C}$ clock cycles. To evaluate an architecture customization, we can use the imaginary architecture as a reference to define its match score, η, with a given problem structure as

$$\eta := \frac{\text{nnz} + L}{\text{nnz} + E_p + E_c L}.$$

The range of η is $(0, 1)$ and can be interpreted as the extra run time introduced in the realized architecture customization, i.e. $T_{\text{img}} = \frac{T_{\text{real}}}{\eta}$. We use this match score to guide the design of problem-specific functional units and data paths. Given a problem structure, we want a customized architecture whose η is close to 1, meaning we want to design

- customized interconnections within the SpMV engine so E_p is close to 0, and
- customized organizations of CVB so E_c is close to 1.

4 PROBLEM-SPECIFIC CUSTOMIZATION

In this section, we present our methodology for optimizing the E_p and E_c of an architecture customization to match the given problem structure. The optimization of E_p is modeled as a string compression problem and and the optimization of E_c is modeled as a mixed integer linear programming problem.

4.1 String Representations of Sparsity Structure

To discuss how to optimize E_p and do connection customization accordingly, we use a string-based encoding to describe the sparsity structure of problem matrices. Each row of the matrix is assigned a character based on the number of non-zero values, transforming the sparsity structure of the matrix into a string. Figure 2(a) is a simple example of such transformation where rows with 1, 2, 3, and 4 non-zero values are assigned a, b, c, and d respectively.

After all the rows are encoded, repeated sub-strings imply similar computation patterns while doing the MV operation. Assuming the non-zero values of the matrices are stored contiguously on the off-chip memory and $C = 4$ elements can be read out at each clock cycle. If there are many ca sub-string, that means there are many rows with 3 non-zero followed by rows with only one non-zero. Consider an 8-input MAC unit with the capability of consuming 4 pairs of inputs to compute their MAC value every clock cycle. To map the ca computation pattern onto the above MAC unit, the rows need to be padded with zeros first to align with the input width, then 2 clock cycles are required to feed in the inputs.

If we allow more flexible connections within the MAC unit like Figure 2(b), the computation throughput of a specific computation pattern can be improved by building a customized reduction tree structure. For the ca pattern, we could instantiate connections like Figure 2(d). Clock cycles of taking the input are reduced to one and the overall computation throughput is improved by 2 times. Given any sub-string which repeats frequently, we could partition the inputs and instantiate connections accordingly to reduce the number of zero-paddings. For convenience, We'll refer to the corresponding MAC unit structure using its associated sub-strings $S = \{s_0, s_1, ...\}$. As an example, Figure 2(c) is a MAC unit with structures $S = \{bb, d\}$. Some connections can be re-used across structures to save circuit areas.

MAC tree structures with different input partitions help improve the computation throughput, but they also lead to variable-length outputs. A companion routing logic is needed to align the outputs with the original data width C for subsequent vector operations as shown in Figure 2(f).

Here we show the simple case of $C = 4$ for illustration purposes and the number of sub-strings patterns is limited. We used up to $C = 64$ in our evaluation and the number of available patterns in the sparsity string encoding becomes much larger. In real problem matrices, we use $\log_2(\text{nnz}_{\text{row}})$ instead of nnz_{row} to encode the sparsity to form a better cluster of similar computation patterns. Rows with less than $1, 2, 4, ..., 64$ non-zero values are mapped to $a, b, c, ..., g$ respectively. If a row has more than 64 non-zero values, we assigned a series of \$ to it and break down the computation to a series of g. Figure 2(g) shows sparsity encoding examples of problem matrices from different applications.

4.2 Optimizing E_p

Our string-based encoding provides a convenient way to discuss E_p optimization. We use the string encoding $dbbaaaca$ of the sparsity structure in Figure 2 as an example to show how to schedule its computations onto the MAC unit with structure $S = \{bb, d\}$ using a series of string replacement.

(1) We first identify all bb in the string and replace them with $*$ (or any character that is not associated with MAC unit structures) as shown in Figure 2(e).
(2) ba, ab, and aa can also be mapped onto bb with some zero paddings and we perform the replacement using a regular expression $ba|ab|aa$.
(3) Finally, we identify and replace d.

The total amount of extra padding can be computed through the length of the final string w_{sched} after the scheduling by:

$$E_p = C \cdot \text{length}(w_{\text{sched}}) - \text{nnz}(A).$$

Note that we search for $bb|ba|ab|aa$ before searching for d. The length of the replaced string and E_p are both smaller in this way. The above scheduling procedure can be generalized to any S and sparsity string encoding. The general rule is that we start the replacement process with the longest sub-string in S and its associated regular expressions. Then we proceed with the second longest sub-string and so on.

In the above example, E_p can be further reduced if we include more structures like ca in S. However, such a reduction comes at the cost of more connections and routing logic for implementing larger $|S|$. This is the major constraint in performing problem-specific MAC tree customization as S can only include a limited number of sub-strings. Let the target size of S to be $|S|_{target}$, then the E_p optimization problem can be formulated as:

$$\begin{aligned} \underset{S=\{s_0,s_1,...\}}{\text{minimize}} \quad & \text{length}(w_{\text{sched}}) \\ \text{subject to} \quad & |S| \leq |S|_{target} \end{aligned} \tag{4}$$

Since the computational complexity of solving Problem 4 is very hard, we used a method based on the dictionary-based lossless compression algorithm LZW [33] to search for a candidate S.

4.3 Optimizing E_c

The vector of the MV operation is stored in CVBs and each buffer has one read port. Due to port limitations, each buffer can only provide random access to one location of the vector at every clock cycle. After the scheduling, C random locations of the vector need to be accessed and multiplied with the matrix non-zero packs at every clock cycle. A simple solution to provide the required parallel access capability is to store C copies of the same vector using C buffers. However, the vector needs to be updated frequently and the simple duplication solution will suffer from slow update time (large $E_c = C$) and also cost many memory resources.

The straightforward duplication of memory organization has many redundancies. Each vector buffer just provides vector elements to one multiplier. After the MAC unit mapping process, only a subset of the vector is scheduled on each multiplier due to the matrix sparsity. Hence, every buffer only needs a partial copy of the vector. We use the matrix sparsity structure in Figure 2(a) as an example. The full copy of the vector has 8 elements: $\{1, 2, 3, 4, 5, 6, 7, 8\}$. The gray numbers in Figure 3(a) indicate that they are never used in the entire computation. After removing the redundant gray copies, the rest of the elements can be compressed to reduce memory usage. The workload of updates and E_c are also reduced. We need 2 clock

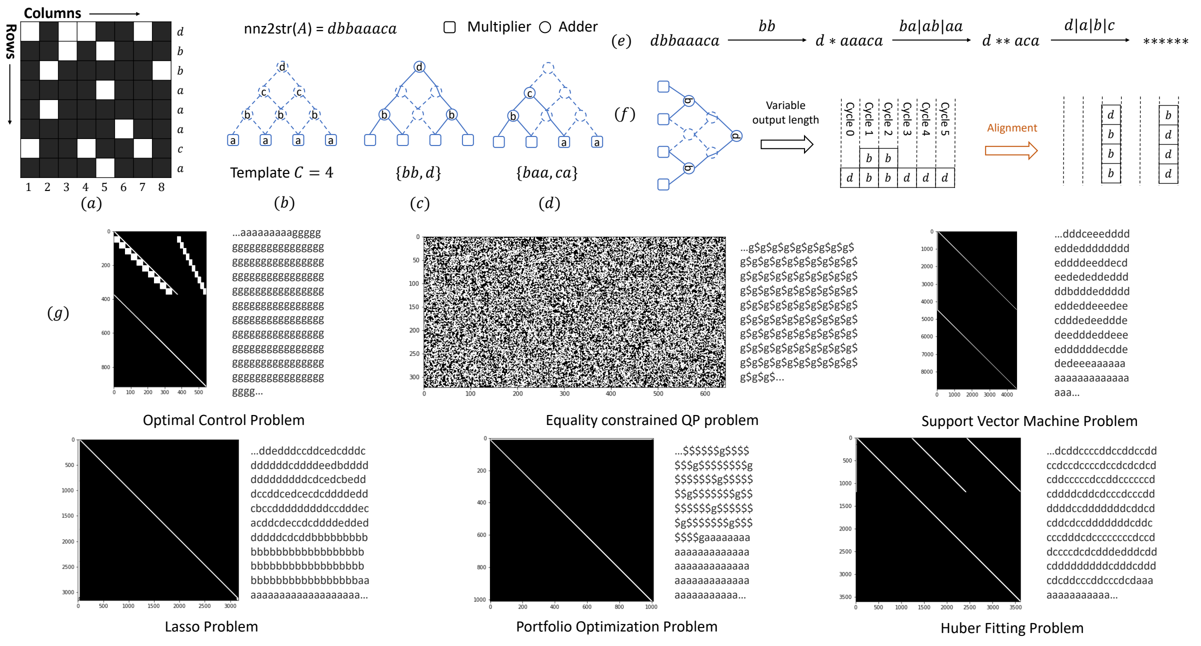

Figure 2: (a)–(f) Problem-specific MAC tree structure design flow. (g) Examples of sparsity encoding from the benchmark problems. The non-zero values of the matrix are represented by white pixels.

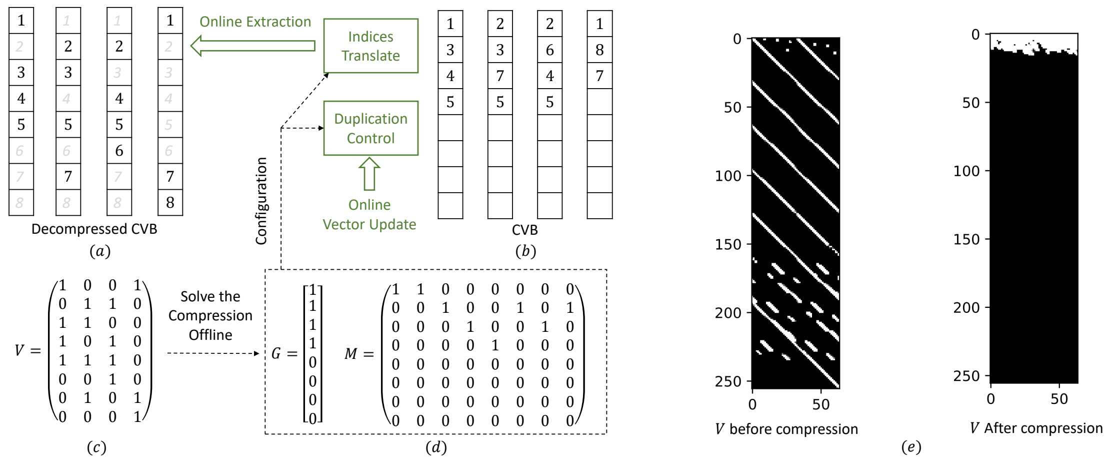

Figure 3: Compressed vector buffers

cycles to update all 1s and 2s in the original memory organization due to memory port limitations. After the compression, we only need 1 clock cycle.

The compression can be done during design time as the sparsity structure of the matrix remains the same across different instances of the problem sequence. Denote the vector access request of each buffer as $V \in \mathbb{R}^{L\times C}, V_{jk} \in \{0,1\}$. $V_{jk} = 1$ means the jth element of the vector is required on buffer k. Denote $M \in \mathbb{R}^{L\times L}, M_{ij} \in \{0,1\}$ as the compression map. $M_{ij} = 1$ means the jth element of the vector can be put at buffer location i. The offline compression can be modeled as a mixed integer programming problem as follows:

$$\begin{aligned} \underset{M}{\text{minimize}} \quad & \sum_{i=0}^{L-1} G_i \\ \text{subject to} \quad & \sum_{j=0}^{L-1} M_{ij}V_{jk} \le 1, \quad \forall i,k, \\ & \sum_{i=0}^{L-1} M_{ij} = 1, \quad \forall j, \\ & \sum_{j=0}^{L-1} M_{ij} \le LG_i, \forall i \end{aligned} \tag{5}$$

The above problem is a mixed integer linear program and is NP-hard. We first modeled the problem using CVXPY [11], but finding the solution to (5) is not tractable, even when $C = 16$ and matrix dimension is 500. Instead, we use the First-Fit algorithm to find an approximate solution. Figure 3(e) shows an example of V before and after compression in an optimal control problem from the benchmark. After the solution to Problem 5 is found, we generate two hardware modules based on M to provide the parallel access capability during run time. The address translation module provides correct copies of the vector to multipliers and the duplication control module updates all vector copies across iterations.

4.4 Adapting the Problem Structure

Note that in the QP (1), we can permute the rows of P and A to alter the problem structure. The solution of the original problem can be recovered through reverse permutation once Algorithm 1 terminates. To optimize E_p, we can permute rows of A to construct a string encoding with more repeat sub-strings, reducing the lower bound of E_p. Row and column permutations can also help to construct a more sparse V in (5), so the achievable compression ratio is higher. However, we must maintain the symmetry of the KKT matrix, so we must permute both the rows and columns by the same sequence. For example, if we choose to swap row i, j of matrix P and row i, j of matrix A^T, the columns i, j of matrix P and columns i, j of matrix A also need to be swapped to maintain the symmetry. Due to this limitation, we observe that adapting the problem structure through matrix row permutation has little improvement effect on the achievable E_p and E_c.

4.5 Architecture Generation

After we solve the E_p and E_c optimization problem, we pass the customization of the MAC tree structure, the indices translation, the duplication map for the CVBs, and the routing logic between the MAC tree and VBs to our hardware generation program for creating the High Level Synthesis (HLS) description of the architecture for the specific problem structure. Figure 6 gives the overview of the generating flow, and Figures 4 and 5 show an example of how the routing logic between the MAC tree and the vector buffer is customized and instantiated for each problem.

5 EVALUATION

In this section, the end-to-end solver performance improvement by applying the problem-specific architecture customization proposed in the previous section is evaluated in detail. The performance and energy efficiency of the customized hardware solver is compared with a GPU implementation of OSQP. In our evaluations, we use the benchmark problems from [30], which contain a wide range of real-world problems from six application domains including portfolio optimization (`portfolio`), Lasso (`lasso`), Huber fitting (`huber`), control engineering (`control`), support vector machines (`svm`), and equality-constrained QP (`eqqp`). There are 120 total problems in the benchmark, and as shown in Figure 7, the number of decision variables ranges from 10^1 to 10^5 and the total number of non-zero values in the P and A matrices of each problem range from 10^2 to 10^6.

```
snippet_file = src_root + 'align_acc_cnt_switch.h'
with open(snippet_file, "w") as HLS_description:
  acc_pack_width = isca_c//char2nnz(arch_code[0], isca_c)
  if len(arch_code)==1:
    HLS_description.write("align_out[0] << acc_pack.data[0];\n")
  else:
    HLS_description.write("switch (acc_cnt) {\n")
    for item in range(len(arch_code)):
      case_width = isca_c//char2nnz(arch_code[item], isca_c)
      HLS_description.write("case {}:\n".format(case_width))
      HLS_description.write("\tswitch (align_ptr){\n")
      for i in range(acc_pack_width):
        HLS_description.write("\tcase {}:\n".format(i))
        for j in range(case_width):
          HLS_description.write("\t\talign_out[{0}] << acc_pack.
    data[{1}];\n".format((j+i)%acc_pack_width,j))
        HLS_description.write("\t\tbreak;\n")
      HLS_description.write("\t}\n")
      HLS_description.write("\tbreak;\n")
    HLS_description.write("}\nalign_ptr += acc_cnt;\n")
```

Figure 4: Problem-specific customization of the routing logic

```
void spmv_align(int align_cnt,
                data_stream align_out[ACC_PACK_NUM],
                cnt_pack_stream &acc_cnt_in,
                data_stream &acc_complete_in,
                spmv_pack_stream &spmv_pack_in)
{
    ap_uint<ALIGN_PTR_BITWIDTH> align_ptr = 0;
align_loop:
    for (int loc = 0; loc < align_cnt; loc++)
    {
#pragma HLS pipeline II = 1
        u16_t acc_cnt = acc_cnt_in.read();
        spmv_pack_t acc_pack;
        if(acc_cnt == CNT_AS_FADD_FLAG){
            acc_pack.data[0]=acc_complete_in.read();
            acc_cnt = 1;
        }
        else{
            acc_pack = spmv_pack_in.read();
        }
        #include "align_acc_cnt_switch.h"
    }
}
```

Figure 5: Instantiating customized routing logic using HLS

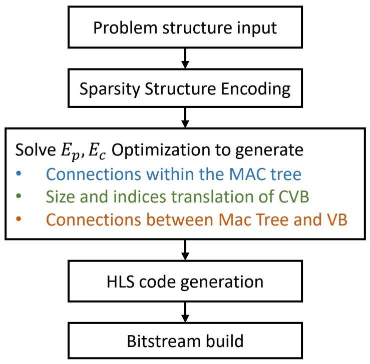
Problem structure input
Sparsity Structure Encoding
Solve E_p, E_c Optimization to generate
Connections within the MAC tree
Size and indices translation of CVB
Connections between Mac Tree and VB
HLS code generation
Bitstream build

Figure 6: Problem-specific hardware generation flow

5.1 Evaluation Setup

The system used for the evaluation contains the devices in Table 2. The U50 FPGA is the major platform for testing different architectures customized for different problem structures, and includes

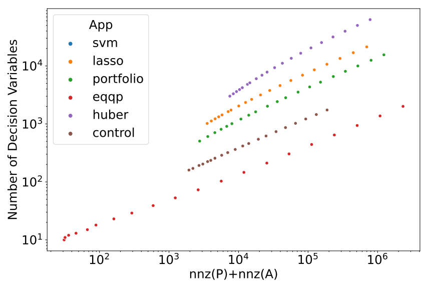

Figure 7: Number of non-zero values in matrices and decision variables in the benchmark

Table 2: Platform Details

Device Model	Peak Throughput	Lithography	TDP
AMD-Xilinx U50 FPGA	0.3 teraflops	16 nm	75 W
Intel i7-10700KF CPU	0.5 teraflops	14 nm	125 W
NVIDIA RTX3070 GPU	20 teraflops	8 nm	220 W

5952 fixed-point DSPs, 28.4 MB on-chip memory, and 8 GB High Bandwidth Memory (HBM). These resources can support all the customized architecture generated for the benchmark. The peak floating-point computation throughput of the FPGA is calculated under the assumption that the clock frequency is 150 MHz and each single precision floating-operation consumes 3 fixed-point DSPs. The GPU used for comparison was the RTX 3070 with NVIDIA's Ampere architecture, which contained 8 GB of off-chip memory and 46 streaming multiprocessors (SMs), each performing a maximum of 128 single precision floating point operations every clock cycle when running at 1.75 GHz.

We used the current 1.0 development branch of OSQP [30] as the benchmark software, with the CPU implementation serving as the performance baseline. This branch can also choose to accelerate Algorithm 2 using either Intel's Math Kernel Library (MKL) [31] or NVIDIA's cuSparse [27] library. The current implementation of RSQP has been integrated with the OSQP software framework, and the U50 FPGA can accelerate the entire PCG computation in the OSQP solver. To switch between the different acceleration backends, OSQP passes a build flag to the compilation process to choose and link the selected backend into the compiled executable binary of the solver. We first run the entire benchmark on MKL using 8 threads. Figure 8 shows the percentage of the solver time spent on Algorithm 2 for each problem in the benchmark. For most problems, solving the KKT system using MKL's CG acceleration takes over 95 % of the total solver run time, showing more efficient acceleration of PCG is necessary to further improve the solver speed.

5.2 Performance Evaluation

We first evaluate the effectiveness of our customization methodology using the metric $\eta \in (0, 1)$ proposed in Section 3. A baseline architecture without applying the E_p and E_c optimization was generated to provide the baseline η for each problem in the benchmark.

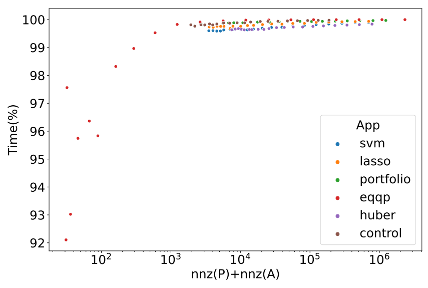

Figure 8: The percentage of the CPU solver time spent on solving the KKT system

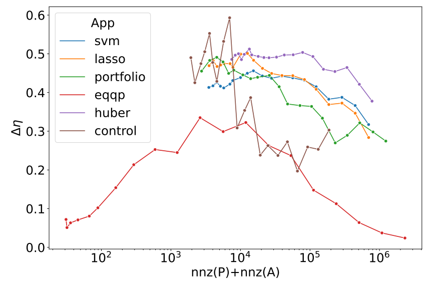

Figure 9: Improvement of η after applying problem-specific customization

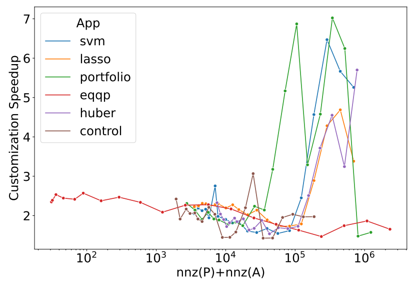

Figure 10: Solver Speedup by Applying Problem-specific Customization

In this baseline, only a single output was used in the MAC tree within the SpMV engine, and C copies of the vector were stored in CVB.

Figure 9 shows the improvement of η after applying problem-specific architectural customization on the benchmark. The results show effective improvements of η on most problems except in the eqqp applications. As shown in Figure 2(g), the matrix sparsity pattern of problems in eqqp is less structural than other applications, which led to the reduced effectiveness in the cutomization process.

Next, we compare the end-to-end solver performance difference between the baseline and the customized architectures. Figure 10

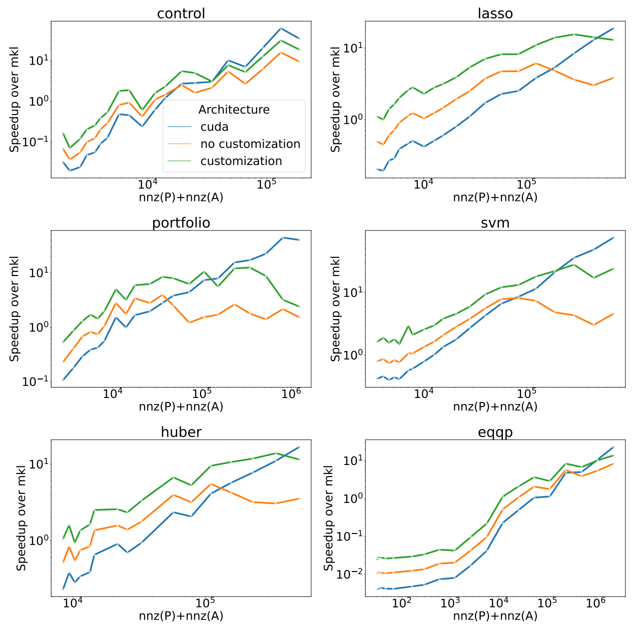

Figure 11: End-to-end solver speedup(higher is better) of FPGA and GPU vs CPU

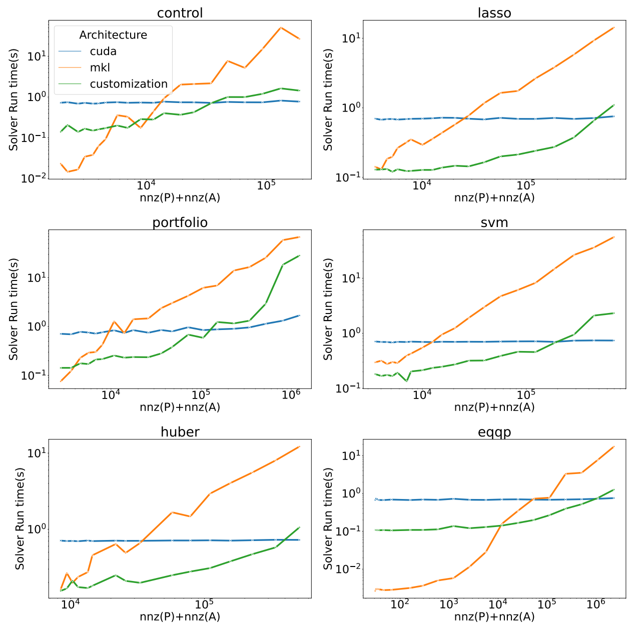

Figure 12: Solver run time (lower is better) on CPU, GPU, and FPGA

shows that our customization methodology achieves 1.4 to 7.0 × improvement. The effect on eqqp is less significant and it agrees with the improvement on η.

Figure 11 shows the speedup over the MKL backend of the baseline architecture by the customized architectures, the baseline architecture before customization, as well as the GPU implementation. As shown in the figure, architecture customization substantially extends the advantage of the FPGA solver to all but the largest benchmark problems. In spite of the much lower theoretical throughput when compared to the CPU and GPU, the FPGA accelerators with architectural customization achieved up to 31.2 and 6.9 times end-to-end speedup, respectively.

5.3 Microarchitectural Performance and Area Trade-off

To study the effect of microarchitectural specialization of the SpMV kernel on performance and resource consumption, a range of design candidates for an instance of the svm problem with 20 616 non-zeros were produced and synthesized for hardware. Results are shown in Table 3. The architectural candidates are denoted as $C\{S\}$ where C is the width of the data path and S is the structure of the MAC unit. For example, $16\{4c1e\}$ denotes an architectural candidate with $C = 16$ and $S = \{cccc, e\}$. Overall, architectures with larger values of C and $|S|$ consumes more hardware and can complete the given SpMV operation in fewer number of cycles. However, the maximum clock frequency that the design may run at ($f_{\max}$) is often affected by the size of the design, causing diminishing return in the final throughput performance. As a result, although architectures such as $64\{64a4e1g\}$ have a big improvement on η through specialization, their hardware frequency are limited by the complex routing logic in the brown part of Figure 1, which ultimately impact their throughput performance.

Table 3: Trade-off Between Performance and Resources

Architecture	$f_{\max}$	$\Delta\eta$	SpMV/μs	DSP	FF	LUT
$16\{e\}$	300	0.000	0.048	80	12218	8556
$16\{16a1e\}$	276	0.226	0.084	80	17190	12502
$32\{32a4d1f\}$	173	0.433	0.130	160	32441	23648
$16\{16a2de1\}$	273	0.345	0.141	80	17350	12623
$64\{64a4e1g\}$	121	0.538	0.144	320	60202	50405
$32\{4d1f\}$	300	0.337	0.150	160	22958	13880
$32\{32a4d2e1f\}$	179	0.484	0.167	160	32581	23812
$32\{4d2e1f\}$	300	0.369	0.172	160	23118	13977
$32\{16b4d1f\}$	257	0.406	0.172	160	27338	17319
$64\{4e1g\}$	270	0.397	0.174	320	42562	23099
$64\{8d4e1g\}$	251	0.489	0.240	320	44403	24245

5.4 Power Efficiency Comparison

To compare the power efficiency of the solvers using the GPU and FPGA backends, we run each benchmark problem 100 times and record the power trace using device management command line tools named `nvidia-smi` and `xbutil`, respectively. The power consumption of the FPGA is steady at 19W while running the benchmark, while the GPU consumes 44W to 126W. Figure 13 compares the energy efficiency, defined as the number of problem instances each device can run using unit power. RSQP can achieve up to 22.7 × improvement over the GPU.

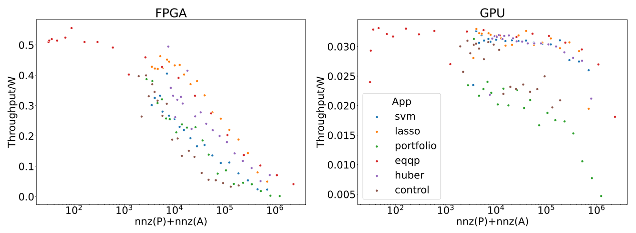

Figure 13: Power efficiency comparison on the benchmark

6 RELATED WORKS

GPU-based implementations of QP solvers have focused on two classes of algorithms: first-order methods (such as ADMM) and interior-point methods. In [18], Huang et. al. implemented an interior-point method for QPs on a single GPU and utilized dense matrices and Gauss-Jordan elimination to solve the KKT equation. Yu et. al. [32] focused on accelerating first-order methods, and showed that writing custom dense matrix-vector multiplication kernels specific for the linear model predictive control (MPC) problem gave a 2.3× speedup compared to using the dense matrix-vector multiplication provided by the cuBLAS library. Recently, Cole et. al. [9] proposed using a problem-level transformation technique that condenses the sparse KKT matrix arising from the interior-point method for the QP (1) into a smaller dense matrix that is then solved using a Cholesky factorization on a single GPU. Compared to a sparse CPU implementation, they reported their dense solver on the GPU reduced the solver runtime by an order of magnitude.

In 2020, Schubiger et. al. [29] proposed the cuOSQP solver, which is an implementation of the OSQP algorithm in the CUDA language. In cuOSQP, the matrices are stored in Compressed Sparse Rows (CSR) format, and then the matrix-vector operation from the cuSparse library is used to perform the computations. cuOSQP was shown to be significantly faster than the CPU-based OSQP for problems with more than 10^5 variables, with the CPU-based implementation performing better for the smaller problems, since the smaller problems cannot fully utilize the GPU or overcome the high initial cost of data movement from the CPU to the GPU.

FPGA implementations of QP solvers have customarily been focused on more specific problem types, and so tend to be specialized to the structure of the P and A matrices in the problem, with the majority of reported solvers being used for the linear MPC problem and using either interior-point methods, active-set methods or first-order methods. These customizations include the use of the Compressed Diagonal Storage (CDS) sparse matrix format for storing block-banded matrices [4] in an efficient manner, and the removal of redundant data (such as repeated matrix blocks, or rows of structured zeros/ones) [22], leading to a memory savings of over 75% for MPC. Compared with the GPU implementations, FPGA QP solvers have been focused on smaller problems with between 10-300 variables found in linear MPC [26], and have reported solver runtimes on the order of 1-10μs [21] and 20-30μs [10] for the solvers tailored to the MPC problem that were hand-coded/optimized in VHDL/Verilog and coded in HLS, respectively.

SpMV acceleration is an important part of our work. There is a rich body of research on related accelerators. [24] proposed a single architecture design for computing both dense matrix-vector multiplication and sparse matrix-vector multiplication. [15] proposed a compressed interleaved sparse rows format to explore parallelism across different matrix rows. [13] extends [15] and avoids the centralized row encoding and decoding. However, they all propose a single acceleration architecture and target matrices with general sparsity structures. One key contribution of our work is the mechanism of matching the architecture configurations with various sparsity patterns. The only similar work we can identify is [17], which proposed an automation flow for generating specific hardware based on the sparsity pattern of the input matrix. However, their framework only works on block diagonal patterns that appear in a specific application domain. As far as we know, we are the first work that describes the sparsity patterns of input matrices in fine granularity and has a systematic approach to generate the architecture accordingly.

7 CONCLUSION

In this paper, we have presented the design and implementation of RSQP, an FPGA accelerated framework for solving convex quadratic programs. The processing architecture of RSQP is customizable, and we developed an automated method to adapt this architecture to the particular sparsity structure of the user input. Using extensive benchmark testing, we showed that significant improvement to the accelerator's performance can be achieved through this architectural customization. The resulting FPGA-accelerated framework also achieves superior performance and power-efficiency over the baseline CPU implementation using MKL as well as an equivalent GPU-accelerated QP solver. Although it takes time to complete the low-level implementation process of a customized FPGA accelerator for a particular optimization problem, this initial hardware creation time is amortized when the same architecture is reused in solving many instances of the same problem with different parameters. By building on the OSQP solver, RSQP can be further integrated with high-level general-purpose optimization frameworks such as CVXPY, opening up new opportunities to accelerate a wide spectrum of real-world parametric convex quadratic optimization problems in the future. We also plan to enhance the user experience through tighter integration between software and hardware to allow seamless hardware generation to be integrated with the high-level user software ecosystem, and to further enhance the performance of the proposed architecture by leveraging improved memory bandwidth in advanced FPGAs.

ACKNOWLEDGMENTS

This research was supported in part by ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR, and the Research Grants Council (RGC) of Hong Kong under project R7003-21. I. McInerney was supported by the Royal Society and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] Karam M. Abughalieh and Shadi G. Alawneh. 2019. A Survey of Parallel Implementations for Model Predictive Control. *IEEE Access* 7 (2019), 34348–34360. https://doi.org/10.1109/ACCESS.2019.2904240

[2] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. 2019. Differentiable Convex Optimization Layers. In *Advances in Neural Information Processing Systems*.

[3] Brandon Amos and J Zico Kolter. 2017. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*. PMLR, 136–145.

[4] David Boland and George A. Constantinides. 2010. Optimising Memory Bandwidth Use for Matrix-Vector Multiplication in Iterative Methods. In *Proceedings of the International Symposium on Applied Reconfigurable Computing*. Elsevier, Bangkok, Thailand, 169–181. https://doi.org/10.1007/978-3-642-12133-3_17

[5] Stephen Boyd, Enzo Busseti, Steve Diamond, Ronald N Kahn, Kwangmoo Koh, Peter Nystrup, Jan Speth, et al. 2017. Multi-period trading via convex optimization. *Foundations and Trends® in Optimization* 3, 1 (2017), 1–76.

[6] Stephen Boyd, Mark T Mueller, Brendan O'Donoghue, Yang Wang, et al. 2013. Performance bounds and suboptimal policies for multi–period investment. *Foundations and Trends® in Optimization* 1, 1 (2013), 1–72.

[7] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press.

[8] Marc-Alexandre Boéchat, Junyi Liu, Helfried Peyrl, Alessandro Zanarini, and Thomas Besselmann. 2013. An architecture for solving quadratic programs with the fast gradient method on a Field Programmable Gate Array. In *21st Mediterranean Conference on Control and Automation*. 1557–1562. https://doi.org/10.1109/MED.2013.6608929

[9] David Cole, Sungho Shin, François Pacaud, Victor M. Zavala, and Mihai Anitescu. 2022. Exploiting GPU/SIMD Architectures for Solving Linear-Quadratic MPC Problems. *arXiv* 2209.13049 (Sept. 2022). https://doi.org/10.48550/arXiv.2209.13049 arXiv:2209.13049 [math]

[10] Aitor del Rio Ruiz and Koldo Basterretxea. 2020. Towards the Development of a CAD Tool for the Implementation of High-Speed Embedded MPCs on FPGAs. In *2020 European Control Conference (ECC)*. 941–947. https://doi.org/10.23919/ECC51009.2020.9143666

[11] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research* 17, 1 (2016), 2909–2913.

[12] Moritz Diehl, Hans Joachim Ferreau, and Niels Haverbeke. 2009. Efficient numerical methods for nonlinear MPC and moving horizon estimation. *Nonlinear model predictive control: towards new challenging applications* (2009), 391–417.

[13] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 54–64.

[14] Yousef Elkurdi, David Fernández, Evgueni Souleimanov, Dennis Giannacopoulos, and Warren J. Gross. 2008. FPGA architecture and implementation of sparse matrix–vector multiplication for the finite element method. *Computer Physics Communications* 178, 8 (2008), 558–570.

[15] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. 2014. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 36–43.

[16] Philip E. Gill, Walter Murray, and Michael A. Saunders. 2005. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Rev.* 47, 1 (Jan. 2005), 99–131. https://doi.org/10.1137/S0036144504446096

[17] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. 2016. Optimising Sparse Matrix Vector multiplication for large scale FEM problems on FPGA. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–9.

[18] Yunlong Huang, Keck Voon Ling, and Simon See. 2011. Solving Quadratic Programming Problems on Graphics Processing Unit. *ASEAN Engineering Journal* 1, 2 (2011), 76–86.

[19] Juan Luis Jerez, George Anthony Constantinides, and Eric C. Kerrigan. 2011. An FPGA Implementation of a Sparse Quadratic Programming Solver for Constrained Predictive Control. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 209–218. https://doi.org/10.1145/1950413.1950454

[20] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. 2013. Embedded Predictive Control on an FPGA using the Fast Gradient Method. In *2013 European Control Conference (ECC)*. 3614–3620. https://doi.org/10.23919/ECC.2013.6669598

[21] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. 2014. Embedded Online Optimization for Model Predictive Control at Megahertz Rates. *IEEE Trans. Automat. Control* 59, 12 (2014), 3238–3251. https://doi.org/10.1109/TAC.2014.2351991 arXiv:1303.1090

[22] Juan L. Jerez, K. V. Ling, George A. Constantinides, and Eric C. Kerrigan. 2012. Model Predictive Control for Deeply Pipelined Field-Programmable Gate Array Implementation: Algorithms and Circuitry. *IET Control Theory and Applications* 6, 8 (2012), 1029–1041. https://doi.org/10.1049/iet-cta.2010.0441

[23] Can B. Kalayci, Okkes Ertenlice, and Mehmet Anil Akbay. 2019. A comprehensive review of deterministic models and applications for mean-variance portfolio optimization. *Expert Systems with Applications* 125 (2019), 345–368.

[24] Srinidhi Kestur, John D Davis, and Eric S Chung. 2012. Towards a universal FPGA matrix-vector multiplication architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 9–16.

[25] Danylo Malyuta, Taylor P. Reynolds, Michael Szmuk, Thomas Lew, Riccardo Bonalli, Marco Pavone, and Behçet Açıkmeşe. 2022. Convex Optimization for Trajectory Generation: A Tutorial on Generating Dynamically Feasible Trajectories Reliably and Efficiently. *IEEE Control Systems Magazine* 42, 5 (Oct. 2022), 40–113. https://doi.org/10.1109/MCS.2022.3187542

[26] Ian McInerney, George A Constantinides, and Eric C Kerrigan. 2018. A survey of the implementation of linear model predictive control on FPGAs. *IFAC-PapersOnLine* 51, 20 (2018), 381–387.

[27] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[28] Peter Nystrup, Stephen Boyd, Erik Lindström, and Henrik Madsen. 2019. Multi-period portfolio selection with drawdown control. *Annals of Operations Research* 282, 1-2 (2019), 245–271.

[29] Michel Schubiger, Goran Banjac, and John Lygeros. 2020. GPU acceleration of ADMM for large-scale quadratic programming. *J. Parallel and Distrib. Comput.* 144 (2020), 55–67.

[30] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. 2020. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation* 12, 4 (2020), 637–672. https://doi.org/10.1007/s12532-020-00179-2

[31] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[32] Leiming Yu, Abraham Goldsmith, and Stefano Di Cairano. 2017. Efficient Convex Optimization on GPUs for Embedded Model Predictive Control. In *GPGPU-10 Proceedings of the General Purpose GPUs*. Austin, TX, US, 12–21. https://doi.org/10.1145/3038228.3038234

[33] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.