Design and Evaluation of a Custom SVM Hardware Accelerator

Taylor Kawate Arunan Elamaran

University of California, Los Angeles | CS 259

Abstract **-** Support Vector Machines (SVMs) are supervised machine learning algorithms that excel at classification tasks by calculating the optimal hyperplane between classes[1]. While SVMs are no longer SoTA for large scale image classification, NLP, RL, and speech recognition tasks after being outclassed by deep learning[2], SVMs still excel at classification for small dataset tasks like text classification and performing inference at the edge[3]. In order to further understand the strengths and weaknesses of different hardware devices performing SVM inference on these small datasets, we design a custom SVM inference hardware accelerator on an FPGA and analyze its performance against equivalent CPU and GPU implementations. By performing design space exploration on the different systems, we found that FPGAs can outperform CPUs and GPUs by multiple orders of magnitude when performing SVM inference for smaller datasets. GPUs excel at processing large datasets, and are better fit for more classical bulk inference tasks, while FPGAs are better fit for smaller tasks at the edge.

I. INTRODUCTION

SVMs excel at small dataset classification tasks, and are often deployed at the edge[3]. For tasks like object detection, image classification, and signal identification, users often care about real-time decision making, requiring low-latency inference capabilities. Depending on the task and resources available, SVM inference can typically be run on either a CPU, GPU, or FPGA.

In this paper, we implemented and optimized SVM inference accelerators on a CPU, GPU, and FPGA to determine when and where each device would excel, with a primary focus on comparing the GPU and FPGA performance tradeoffs. These two devices are most commonly used for ML related inference acceleration, and have fundamentally different hardware structures, making them an interesting pair to separately optimize for and compare their performance.

We found that FPGAs are highly efficient at performing SVM inference for small datasets, and can achieve very low latencies, even on cheaper, smaller FPGAs. This is due to their simple, customizable structure capable of being fine tuned to accelerate inference for a specific dataset and operation. On the other hand, GPUs have a much larger overhead, but their highly parallel structure and higher throughput makes them optimal for handling large inference tasks processing loads of data.

II. BACKGROUND

A. Support Vector Machines (SVMs)

SVMs are supervised machine learning algorithms that can find the optimal hyperplane between classes, making them very good at classification tasks. SVMs maximize the margin between the support vectors (closest points to the hyperplane) in order to best separate classes. SVMs can operate on two or more classes, though we focus primarily on binary classification for our proof of concept design space exploration. They can also handle multiple features, and use them to identify the best possible hyperplane between classes. This is often done using kernel tricks to map data into higher dimensions until the data becomes separable. Lastly, while we focus on linearly separable classes, SVMs can also find non-linear decision boundaries, even with multiple classes[1].

Like many other machine learning algorithms, SVMs have two main phases: training and inference. In training, the model learns the optimal hyperplane best separating the classes of data based on training data points. Weights are tuned during this phase, and are used to predict the class of the data point. This predicted class (label) is compared against the actual training label, which is used as feedback in tuning the model's weights. Once trained, SVM models can perform inference, where unseen test data points are fed forward through the model's pretrained weights, outputting a predicted label. This label is compared against the actual test point's label, allowing the test accuracy to be computed [4].

We implemented two versions of SVM inference on the CPU, each corresponding to a different mathematical formulation of the SVM decision function: the dual form and the primal form. While both achieve the same end goal—predicting the class of a given input vector—they differ in how the model is represented and what data is required at inference time.

The first implementation uses the primal form of the linear SVM decision function, where the decision boundary is represented directly by a weight vector:

$$f(x)=\text{transfer}(w\cdot x+b)$$

- w: the weight vector, precomputed during training as $\Sigma a_i y_i x_i$; it encodes the net influence of all support vectors
- x: the input vector to classify
- w·x: the dot product measuring projection of input onto the weight vector
- b: the bias term, same as in the dual form

This version is computationally efficient at inference time because it avoids iterating over support vectors—it requires only a single dot product between the input and the fixed weight vector. However, it assumes that w has been explicitly computed and stored.

The second implementation corresponds to the standard dual formulation of a linear SVM. The function is:

$$f(x) = transfer(\Sigma_{i=1}^{N} a_i y_i (x_i \cdot x) + b)$$

- x: the input vector to classify
- xi: the x_ith support vector (a training point lying closest to the decision boundary)
- ai: the learned dual coefficient (Lagrange multiplier) for the support vector xi; represents the importance (or weight) of that vector in the decision
- yi: the label of xi, encoded as -1 or +1
- xi * x: the dot product between support vector and input vector; measures alignment (similarity) in input space
- b: the learned bias term, which shifts the decision boundary

This implementation requires storing and looping through all support vectors at inference time. It is directly derived from the way SVMs are trained using kernel methods and is suitable when the number of support vectors is manageable.

B. Related Works

Catanzaro et al. (2008) presented one of the earliest works on GPU-accelerated SVMs, titled "Fast Support Vector Machine Training and Classification on Graphics Processors". Their implementation leveraged CUDA to parallelize the computationally intensive portions of the Sequential Minimal Optimization (SMO) algorithm used during training. Specifically, they optimized the kernel matrix computations—normally a bottleneck—by mapping dot products and kernel evaluations to thousands of GPU threads. This allowed them to achieve up to 100x speedups compared to CPU-based LIBSVM on some datasets. However, they acknowledged that GPU memory limitations restricted scalability for large SVM models due to the need to store the full kernel matrix in memory [11].

cuML, part of NVIDIA's RAPIDS suite, includes modern GPU-accelerated implementations of SVMs for both training and inference, written in CUDA and accessible via Python. It provides both linear and kernel SVMs and integrates with other RAPIDS components for end-to-end GPU data science workflows. The cuML SVM implementation utilizes high-level CUDA libraries like cuBLAS, CUB, and Thrust to achieve efficient memory access and computation. Unlike earlier works focused solely on raw speedups, cuML prioritizes usability, scalability, and integration with dataframes and DLPack tensors. This makes it suitable for real-world applications involving large datasets and diverse ML pipelines [12].

Many previous papers have been dedicated to SVM acceleration on FPGAs[4], exploiting many different architecture options. These papers largely focus on balancing both latency and throughput through parallelism and pipelining. They also handle the main computation, multiplication, either with CORDIC algorithms, DSP slices, or with shifters and adders. However, these papers all generally focus on high end FPGAs and achieve speeds typically between 50MHz and 250MHz. For this paper, we wanted to focus on a smaller FPGA and take advantage of its small size and resources to fully optimize the design, proving that even a $120 FPGA could compete with higher end FPGAs and GPUs through clever architecture techniques and design decisions.

III. METHODS

A. SVM Training

Before implementing the SVM inference cores on our GPU and FPGA, we needed to first obtain a dataset that met our requirements so we could train our model and produce our weights. As mentioned previously, SVMs are primarily useful when classifying small datasets, not large ones due to the intensive computations required for classifying each data point. We wanted a small dataset that would be flexible and simple to work with, one that had linearly separable classes but also could be scaled if necessary to non-linear, multi-class SVMs.

The Iris dataset from SciKit[5] met these requirements. This dataset provides 50 data points per class, with three classes of iris flowers: setosa, versicolor, and virginica irises. Each class had four features: sepal length, sepal width, petal length, and petal width.

For our binary linear SVM classifier, the setosa and versicolor data points are conveniently linearly separable using just the sepal length and width features (Figure 1). We split the corresponding 100 data points for these classes into a 70-30 train test split, meaning 35 train points from each class, and 15 test points from each class. These two classes' training points were used to train our main SVM model, which was a regular SVM model from SciKit[6]. After training the model, we verified its accuracy by plotting against the test data (Figure 2), and printed its accuracy (Figure 3).

SVM_complex.ipynb was then written, still performing binary classification on the setosa and versicolor classes using only the first two features (sepal length and width), but its purpose is different: it explicitly retrieves the learned support vectors, alpha coefficients (Lagrange multipliers), labels (converted to ±1), and bias term from a linear SVM trained with SciKit's SVC class. While implementing a non-linear kernel function (K) was considered, we deemed that doing so would not reveal additional essential information that a linear kernel would not.

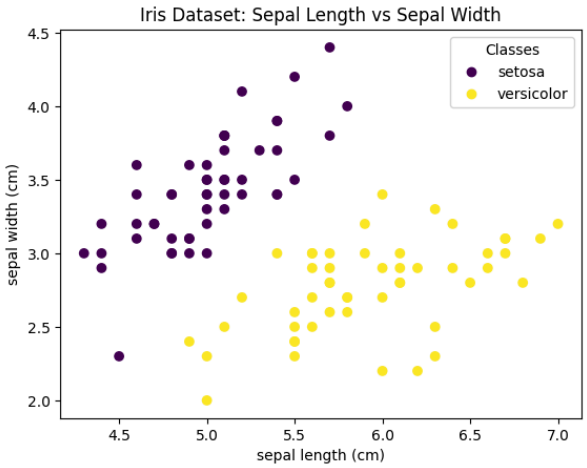

Figure 1: Linearly separable iris classes based on sepal length and width.

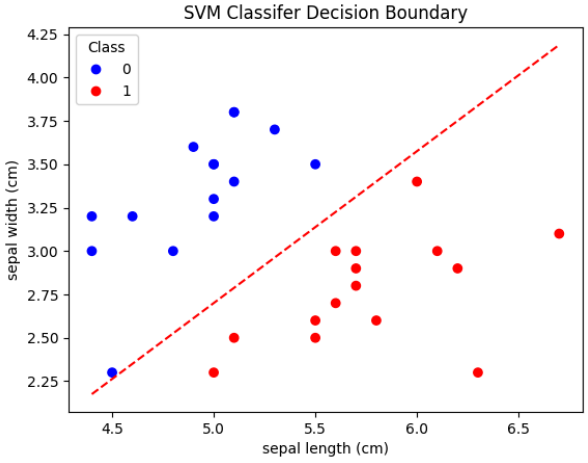

Figure 2: Linearly separable iris classes based on sepal length and width, divided by the decision boundary.

```
# Print accuracy
accuracy = clf.score(test_data, test_labels)
print(f"Testing accuracy: {accuracy:.3f}")

Testing accuracy: 1.000
```

Figure 3: 100% accuracy on the test data.

Once the decision boundary was learned, we then exported the weights so they could be used in our GPU and FPGA inference accelerators.

B. CPU Inference

In the primal formulation of SVM inference, each input vector is classified by computing a dot product with a fixed weight vector, followed by the addition of a bias term and the application of a transfer function. On the CPU, this computation is expressed as a nested loop over inputs and their feature dimensions, forming a simple yet structured access pattern. Conceptually, this setup creates several opportunities for efficient data access. The weight vector, which remains constant across all inputs, is accessed repeatedly with the same stride, exhibiting strong temporal reuse. Meanwhile, input vectors are stored in a linear, row-major layout, allowing sequential access across both the input and feature dimensions. This results in favorable spatial locality, which modern CPU caches can exploit to reduce latency and improve throughput.

Although the CPU implementation is scalar and sequential, its memory access pattern and nested loop structure make it amenable to tiling or batching strategies, particularly in the context of cache-aware optimization or when transitioning to hardware like GPUs. For instance, blocking the input batch into chunks that fit within cache or shared memory would allow multiple input vectors to be processed without evicting frequently reused data. Additionally, the regularity of the dot product calculation enables loop unrolling or vectorized execution in both software (e.g., using SIMD instructions) and hardware. While the dimensionality in common datasets may be small (e.g., two or four), the structure generalizes to higher dimensions and lends itself to memory coalescing and shared memory reuse in massively parallel environments.

The dual-form implementation of SVM inference computes the classification score for each input vector by accumulating contributions from all support vectors, where each contribution is based on the dot product between the input and a support vector, scaled by the corresponding alpha coefficient and label. This nested loop structure—iterating over support vectors and feature dimensions for every input—introduces a natural pattern of data access that reveals several optimization opportunities from a memory hierarchy standpoint. Most notably, the input vector for a given iteration is reused repeatedly across the inner loop over support vectors, which results in temporal locality for that input data. This reuse can be captured on CPUs by ensuring the input vector remains in cache during the processing of all support vectors, reducing redundant memory fetches. In the innermost loop, each feature dimension of the input and support vectors is accessed in a stride-one fashion due to the row-major layout, offering strong spatial locality within the dot product computation itself.

The support vectors, in contrast, are traversed sequentially across inputs, meaning that each support vector is accessed once per input but is reused across the entire input batch. This global read-only reuse of the support vector matrix suggests potential for tiling or blocking in the support vector dimension. For example, when input batch sizes are large, the support vectors can be loaded into a smaller working set—such as a scratchpad or shared memory region—and reused across a tile of input vectors before being evicted. This kind of batched tiling reduces memory traffic and allows more efficient use of fast memory hierarchies. Moreover, since the alphas and labels are small, one-dimensional arrays accessed linearly and repeatedly, they benefit from high cache hit rates and are amenable to constant memory usage in a parallel architecture.

Despite its nested structure and apparent computational overhead, the dual form exposes a high degree of data regularity and loop-level reuse that maps naturally to

parallelization strategies in both software and hardware. The dot product loop itself is a good candidate for unrolling or vectorization, and the overall input loop could be batched to take advantage of multithreaded SIMD units on modern CPUs. Structurally, this CPU version reveals a layered memory access pattern—with input-local reuse, support vector reuse across inputs, and consistent indexing—that can be exploited in more advanced accelerator designs, including GPUs with shared memory tiling or memory-coalesced thread mappings.

Beyond its algorithmic structure, the CPU-based SVM inference implementations—both primal and dual—offer several opportunities and caveats at the microarchitectural level that affect performance. One of the most significant is the way data access patterns interact with the cache hierarchy. Since both input and support vectors are stored in row-major format, accesses to features across dimensions occur in a stride-one fashion during dot product calculations. This alignment enables highly efficient use of cache lines and allows modern CPUs to leverage hardware prefetchers, which can fetch multiple cache lines in advance of use, hiding memory latency and boosting instruction throughput. This benefit is more pronounced when input vectors are processed in large, contiguous batches.

In the dual form especially, where each input vector is reused across a full loop over support vectors, temporal locality becomes important. Retaining the currently processed input vector in the L1 or L2 cache during all support vector iterations avoids repeated fetches from main memory. Conversely, the support vectors themselves exhibit read-only reuse across the entire input batch, but less temporal locality within the inner loops. As a result, blocking techniques could be applied to bring subsets of support vectors into faster memory regions and reuse them across multiple input vectors in tile-sized batches, maximizing the effective bandwidth.

Another low-level concern is false sharing, which may arise in multithreaded CPU implementations. When adjacent entries in the outputs array are written to by separate threads, they may occupy the same cache line, leading to invalidation and coherence traffic. Padding output entries or aligning them to cache line boundaries can mitigate this problem. Furthermore, the regularity and tight inner loops in dot product computations make them strong candidates for vectorization via SIMD instructions. Compilers can often unroll and fuse loops automatically, but performance can be further improved through manual loop unrolling when dimensionality is fixed or small (e.g., 2D or 4D).

Finally, the dual form tends to be memory-bound due to the ratio of floating-point operations to memory accesses. This means performance is often limited not by compute throughput, but by how quickly data can be moved between cache and registers. Techniques such as software pipelining or loop reordering can sometimes mitigate these bottlenecks, and cache-aware memory layouts (e.g., structure-of-arrays over array-of-structures) can also improve memory utilization. The simple scalar form of the CPU loop hides these opportunities, but they become critical when optimizing for modern multi-issue, superscalar CPUs or when preparing for GPU acceleration.

C. GPU Inference

For the GPU implementation of linear SVM inference using the primal form, we explored multiple conceptual strategies for how the workload could be distributed across the GPU's threads and blocks. The primal form of SVM simplifies inference by reducing the classification decision to a dot product between a precomputed weight vector and each input vector, followed by the addition of a bias term and application of a transfer function. Because each input vector can be processed independently, this formulation is inherently parallelizable, but the granularity and mapping of work across threads and blocks significantly affect both performance and scalability.

One intuitive and highly scalable strategy is to assign one thread per input vector, where each thread independently computes the dot product between a single input and the shared weight vector. This thread-parallel approach is conceptually simple and scales efficiently with batch size, making it well-suited for high-throughput applications involving a large number of input vectors. Since each thread performs all necessary computations for one input, there is no need for intra-thread communication or synchronization, and memory accesses for input vectors can be coalesced across threads in a warp. Furthermore, because the weight vector is common to all threads and remains fixed across inference, it can be stored in constant memory. This allows threads to access it rapidly with broadcast efficiency, especially when the model dimension is small. In the special case of two-dimensional input vectors, the dot product can be manually unrolled for additional efficiency, avoiding loops and reducing register usage. This approach is ideal for small, fixed-size problems where high occupancy and low latency are paramount.

A second conceptual approach explores block-level parallelism, where a single thread block is assigned to process one input vector in parallel. In this model, threads within the block collaborate to compute the dot product between the input and weight vector, dividing the dimension of the input across threads. Each thread is responsible for computing a partial product over a subset of dimensions, and the partial results are accumulated into shared memory. A tree-based parallel reduction is then performed to sum these partial results efficiently, allowing the final decision value to be computed and written by a single thread. This design is beneficial when the input vector dimension is large enough to justify intra-block parallelism or when minimizing per-thread register pressure is important. It also enables efficient reuse of the input vector and weight vector within the block. However, the main limitation of this approach is scalability. Since each block processes only one input vector, the number of concurrently active blocks is bounded by the total number of inputs, which can limit GPU occupancy on small datasets. Additionally, the use of shared memory and synchronization incurs extra overhead, which may not be justified for low-dimensional data.

These two strategies—thread-parallel and block-parallel—offer complementary trade-offs. The thread-parallel model offers superior scalability and simplicity, particularly for small or

moderate input dimensions, and excels in scenarios where the number of input vectors dominates the problem size. The block-parallel model, while more complex, offers potential performance gains for very high-dimensional inputs, where splitting the dot product computation across threads can reduce individual thread load and better utilize shared memory bandwidth. Conceptually, both approaches share a reliance on the fixed, precomputed weight vector and exploit the independence of input vectors to avoid synchronization across blocks. Additionally, both strategies can benefit from kernel specialization: hardcoding the dimensionality of the inputs (e.g., 2D) allows for loop unrolling and constant memory use, further improving efficiency for known workloads.

In contrast to the primal formulation, which uses a single weight vector to compute classification scores, the dual form of SVM inference expresses the decision function as a weighted sum of dot products between an input vector and a subset of training points known as support vectors. Each support vector contributes to the classification decision according to its learned alpha coefficient and original class label. As a result, the inference process in the dual form requires computing the dot product between each input and every support vector, scaling that result by the corresponding alpha and label, and summing over all support vectors. This formulation is computationally more intensive than the primal form but offers flexibility, especially in kernelized or sparse models.

Given this structure, multiple GPU parallelization strategies can be considered, each reflecting a different mapping of work across threads and blocks. One straightforward approach is to assign a full thread block to each input vector, where each thread within the block handles one support vector. This model allows the dot product computations for a single input vector to be distributed in parallel across support vectors, with threads accumulating their partial results into shared memory. A tree-based reduction is then used to sum these contributions into the final classification score. This block-per-input strategy is conceptually desirable for low- to moderate-sized inputs, as it enables high intra-block reuse and allows the support vector set to remain constant across the block. However, its scalability is constrained by the number of inputs. If the input batch size exceeds the number of thread blocks that can run concurrently on the GPU, overall throughput can suffer. Furthermore, it introduces synchronization overhead and relies on a shared memory buffer, which can become a limiting factor as support vector count or dimensionality grows.

Another approach, more naturally scalable, assigns one thread per input vector, with each thread independently computing the full decision function across all support vectors. This thread-per-input model benefits from simplicity and efficient scaling to large input batches. However, it introduces significant global memory traffic, as each thread redundantly accesses the full support vector matrix. To address this, shared memory can be leveraged: support vectors are loaded cooperatively into shared memory once per block, after which threads within the block reuse the data for their respective inputs. This technique reduces bandwidth demands and accelerates computation, particularly when the number of support vectors is moderate and fits comfortably within shared memory. The cost of loading into shared memory is amortized over all threads in the block, making this model well-suited for practical GPU architectures.

At the other end of the design space is a fine-grained approach where each thread handles the dot product between a single input and a single support vector. In this case, the decision function is constructed through a two-dimensional mapping of threads: one dimension iterates over inputs, and the other over support vectors. Each thread computes a partial contribution to a classification score, and reductions are required both across support vectors and across threads working on the same input. While this strategy maximizes parallelism, it comes with high coordination and synchronization costs and is rarely efficient unless the problem size is extremely large or very sparse.

All of these strategies can also be adapted depending on whether the dimensionality is known at compile time. When working with low-dimensional input spaces (such as 2D classification tasks), dot product computations can be unrolled manually, and constant memory can be used to store support vector values for fast, broadcast-style access. This reduces register pressure and loop overhead, making it ideal for latency-sensitive inference on small models.

When mapping SVM inference kernels to the GPU, several low-level hardware considerations play a crucial role in performance. Chief among these is the layout of memory and the way it aligns with the GPU's warp-based execution model. In the thread-per-input design, each thread independently processes one input vector, meaning that neighboring threads often access consecutive memory addresses within the input_vectors array. If the data is properly aligned and stored in row-major format, these accesses will be coalesced—i.e., grouped into single memory transactions per warp—maximizing global memory throughput. However, if input vectors are misaligned or padded incorrectly, memory accesses become strided or fragmented, causing multiple memory transactions per warp and significantly reducing effective bandwidth.

In the dual-form GPU kernels, especially the thread-parallel and block-parallel designs, shared memory becomes a powerful optimization tool. For example, loading the support vectors into shared memory cooperatively allows threads to reuse these values many times without repeatedly accessing global memory. This shared memory reuse is particularly effective when the number of support vectors and the dimensionality are small enough to fit within the block's shared memory allocation. However, storing support vectors in shared memory introduces the possibility of bank conflicts. Shared memory is divided into multiple banks, and if threads within a warp access addresses that map to the same bank, serialization occurs. To avoid this, padding the shared memory layout or restructuring access patterns to spread loads across banks ensures bank conflict-free access and preserves warp-level efficiency.

Another essential performance consideration is occupancy versus resource usage. For example, kernels that rely heavily on shared memory—such as those that preload full support vector matrices—may reduce the number of concurrent thread

blocks that can be scheduled on a Streaming Multiprocessor (SM). While fewer active blocks might lower occupancy, the overall performance may still improve if the active threads are doing more useful work per cycle and making better use of on-chip resources. Conversely, lightweight kernels with low register and memory pressure can achieve higher occupancy but might underutilize compute or memory bandwidth. Balancing these trade-offs is critical and often requires empirical tuning of thread block sizes, shared memory usage, and register allocation.

In reduction-based block-parallel kernels, where threads within a block compute partial sums of a dot product and accumulate results in shared memory, careful implementation of tree-based reductions is essential. Synchronization via __syncthreads() must be placed correctly between reduction steps to avoid data hazards. Furthermore, as reduction depth increases (particularly with wide blocks), divergence may become more prominent, which can hurt warp efficiency. Structuring reduction code to maintain contiguous and aligned memory accesses minimizes this effect.

Finally, kernel launch configuration plays a nontrivial role. Thread block sizes should ideally be multiples of the warp size (usually 32) to avoid underutilization of warps. Moreover, when using dynamically allocated shared memory, the size must be carefully bounded so as not to exceed per-block limits imposed by the architecture, which would cause launch-time failures or fallback to lower-performing configurations.

D. FPGA Inference

We used an Altera DE-10 Lite 10M50DAF484C7G FPGA for this paper. Despite its small size and low cost, this FPGA contains 50K 4-input LUTs, 144 18-bit multipliers, 1,638Kbits of M9K embedded SRAM memory, and 4 PLLs, among other features[7].
The SVM inference operation ultimately involves four main phases that can be optimized for when implemented on an FPGA: data retrieval, multiplication and accumulation (MAC), classification, and data storage.

Data Retrieval: as a reminder, we are working with two classes, each with two features, and 15 samples per class (60 values). We also have three weights: w0, w1, and b.
Data can either be stored in memory, or streamed in from the FPGA's GPIO pins. For this particular experiment, we focused on storing data in memory as this is more scalable than streaming in data, especially when processing slightly larger datasets.
When storing data in memory, data can be stored in the FPGA's M9K embedded SRAMs, off-chip DRAM, or even hardcoded into the LUTs. Between these options, we opted to store our inputs in the embedded SRAMs because the access times would be way faster than DRAM, and because the data inputs and weights could actually fit in SRAM.
In SRAM, data could either be distributed to parallel M9K units and read out serially, or written to fewer M9K units and bit selected to read in parallel instead (Figure 4) (the bit selection is a logic technique that would allow us to bypass the single or dual read limitations per cycle from SRAM). However, the former was ultimately chosen for storing the data inputs since while it does use more M9K units, it also allows each M9K unit to be routed closer to the next stage logic, and reduces long routing congestion, a major source of delays in FPGA fabrics.

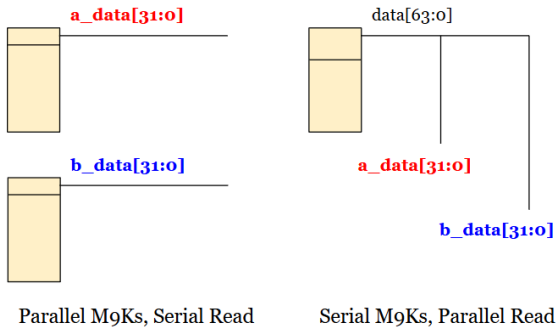

Figure 4: M9K data orientation options.

The M9K units could be configured in multiple ways, but we inferred the single-port ROM configuration in our RTL[9]. Single port was chosen over dual-port to reduce logic complexity and increase access times, as SRAM ports are very expensive.
Identifying the best way of storing the weights was a bit less clear, as the weights were reused in all parallel computations, unlike the inputs. Storing the weights in one ROM would be the most space efficient, and bit selection could be used to read all three values out of one ROM in one cycle. However, the routing would once again be a major concern. The main remaining options would either be to duplicate the weights and store them in multiple ROMs, or to use LUTs to hardcode the constants. This tradeoff is quite convoluted, so we opted to just hardcode the constant in the RTL and let the compiler infer the best resource to use.

MAC: two multiplications per input sample are required for our SVM core, as a result of having two features (w0*x0 and w1*x1). Multiplications are very computationally expensive, hence why FPGAs have dedicated DSP units to take care of these operations. However, as mentioned in related works, sometimes multiplication on DSPs is replaced by using the iterative CORDIC algorithm[10], or by using shifts and adds[10]. However, in this case we have 18-bit multipliers on-die, and they are very close in proximity to the M9Ks (Figure 5).

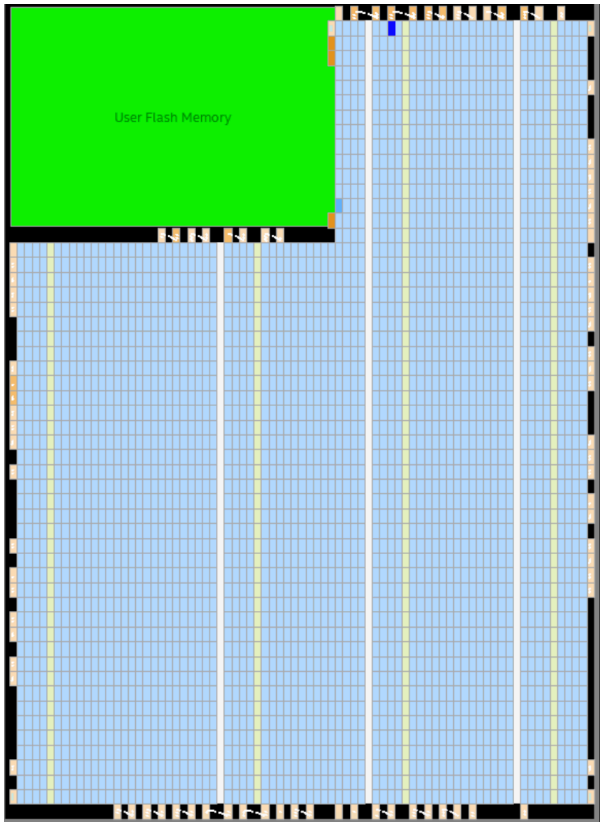

Figure 5: 10M50DAF484C7G FPGA die. The yellow rectangles are M9K units, and the white rectangles are the 18-bit multipliers, both of which are oriented in vertical strips.

CORDIC is an iterative process that inherently has a very high latency, and also requires multiple LUTs (as opposed to just using two multipliers per sample). Similarly, though we can use the shift and add multiplication here as we are multiplying by known constants, doing so requires many LUTs and would likely underperform the dedicated multiply units. As a result, we opted to use the dedicated multiply units, and aimed to use 18-bits of precision (or less) for our multiplications in order to optimally fit to our hardware.
These products are then summed together along with the bias weight. Unlike multiplication, addition is typically performed on LUTs, though some FPGAs have dedicated carry chains for additions as well. Here, we once again let the compiler infer what resource would be most optimal for addition.

Classification: at the end of the MAC computations, each sample is classified as one class or the other depending on if the resulting value is greater than or equal to 0. However, in our design we use 2s complement number representations. Doing so lets the compiler properly handle all sign differences as we deal with positive and negative numbers in SVM. But importantly, it also makes sign detection easy, as the MSB of all 2s complement numbers inherently represent the sign of the value. As a result, there is no need for comparators, and we can simply sign select the resulting value and use it to select one class or the other. This sign selection requires essentially no overhead, and is very optimal both temporally and spatially.

Data Storage: after all the inference is complete, the data needs to be stored somewhere in order to be read out for practical purposes. Once again, this data can be streamed through GPIO pins or stored in memory, and we chose to store it in memory for the same reason as before. However, unlike the input storage, we concatenate the outputs together and write them into fewer M9Ks (one in this case). This is done because the output labels are only 1-bit wide, and because reading out the data from one M9K makes viewing the results of the inference much easier. Though there is increased routing congestion when fanning-in to one M9K, it practically makes sense to write all output data to one place.

Other Optimizations: beyond the aforementioned optimizations and design choices, there are a few other areas we optimized for our FPGA design, briefly mentioned below:

- **Pin Planning**: routing the top level signals to the closest possible pins on the FPGA helps reduce interconnect delays.
- **SDC Timing Constraints**: placing tighter timing constraints on the design helps the place and route (P&R) tool more efficiently constrain the design and choose the optimal placement of resources.
- **PLLs**: our FPGA only runs at a max native frequency of 50MHz. However, by feeding the reference 10MHz clock into a PLL, we can output and run on much faster clock frequencies. As will be explained later, we use a PLL to run on the hardware at 250MHz (though if we had better hardware, our optimized design could run much faster).
- **Precision**: using lower precision inputs and weights has the advantage of being less resource intensive, at the cost of lower precision potentially impacting accuracy. Precision also directly affects our LUT, multiplier, and ROM usage and utilization. As a result, we used 18-bit fixed point precision (6-bit integers, 12-bit fractions) for our "high precision" design, which would perfectly fit in our 18-bit multipliers. We also tested a lower precision version as well, using 7-bit precision (5-bit integers, 2-bit fractions).
- **Pipelining**: since our focus was on low latency, our main design implementation used minimal pipelining and maximum parallelism in order to process all input data in one cycle at very low latency. However, as a proof of concept we also implemented a pipelined version for a more throughput-oriented design.
- **Scripts**: though not a hardware performance optimization, it is worth noting that shell scripts and python scripts were used in unison to help create ROM-inferred RTL, convert weights and inputs to the proper precision and format, and print out data to make writing the RTL easier and faster.

IV. METHODOLOGY

A. Primal Form Implementation (CPU & GPU)

Two CPU inference functions are provided. The first, svm_inference_batched_cpu_general, handles input vectors of

arbitrary dimensionality. It loops over each input, computes the dot product with the fixed weight vector, adds the bias, and stores the result after applying a transfer function. The inner loop performs scalar multiplications and accumulations across the dim dimensions. This version is flexible but slightly less efficient for low-dimensional data.

To improve performance for 2D inputs, a second specialized function, svm_inference_batched_cpu_2d, was implemented. This version avoids looping and explicitly unrolls the dot product, using hardcoded indexing for the two input dimensions. This reduces branching, improves instruction-level parallelism, and enables easier compiler optimization.

Three GPU kernels are provided to explore different parallelization strategies, all based on the primal form.

In thread-parallel mode, each thread is assigned one input vector. The general version, svm_inference_threaded_kernel_general, supports arbitrary dimensions and accesses the weight vector from global memory. Each thread computes the full dot product, adds the bias, and writes the result to global memory. To optimize performance for small input dimensions (specifically 2D), a specialized kernel, svm_inference_threaded_kernel_2d, was created. This kernel loads the weight vector into constant memory (__constant__) and manually unrolls the dot product computation, eliminating loops and reducing register usage. This results in more efficient execution, especially on warp-level hardware.

In block-parallel mode, the kernel svm_inference_batched_kernel assigns one thread block to each input vector. Threads within a block compute partial dot products over disjoint chunks of the feature dimension, accumulating partial sums into shared memory. These partial results are then reduced via a parallel tree-based reduction. Only thread 0 writes the final result. This approach is well-suited for high-dimensional input vectors where a single thread would not fully utilize GPU compute resources. By distributing the workload across threads and using shared memory for intermediate values, this kernel reduces register pressure and takes advantage of on-chip fast memory. Synchronization barriers (__syncthreads()) ensure correctness during both the dot product and the reduction stages.

The adjustable parameters for testing are the following:

- DIM specifies the dimensionality of input vectors and weight vectors. If set to 2, the implementation uses the specialized fast path with loop unrolling and constant memory. If greater, the general path is used.
- NUM_INPUTS determines the number of input vectors processed in the batch. This affects both total workload and memory allocation size. It also affects kernel grid size.
- THREADS_MAX caps the number of threads per block in either mode. This helps control register and shared memory usage to fit within hardware limits.
- BLOCK_THREADS defines the number of threads per block for the block-parallel kernel. It must balance shared memory pressure (one float per thread) with sufficient parallelism to make reduction worthwhile.
- THREAD_THREADS specifies threads per block in thread-parallel mode. It is usually set to the lesser of THREADS_MAX and NUM_INPUTS to maximize warp efficiency while avoiding over-provisioning.
- RUN_MODE selects between THREAD_PARALLEL and BLOCK_PARALLEL execution. This switch controls which kernel path the GPU dispatcher will invoke.

Together, these parameters provide fine-grained control over the memory layout, compute intensity, and resource usage of each kernel variant, allowing the implementation to be tuned to specific input sizes, dimensions, and target GPU architectures.

B. Dual Form Implementation (CPU & GPU)

The function svm_inference_batched_cpu serves as the reference implementation. It iterates over all input vectors and for each one, loops over all support vectors to compute the dot product between the input and support vector. The result is then scaled by the corresponding alpha and label and accumulated into a running sum. After processing all support vectors, the bias is added, and a transfer function is applied to produce the final output. All memory accesses are sequential and cache-friendly, and the nested structure makes it clear how the algorithm maps to the dual form SVM equation.

The GPU implementation offers three kernel variants: SINGLE, BLOCK, and THREAD, each capturing a different mapping of computational work onto the GPU's thread hierarchy.

In the SINGLE mode, the kernel svm_inference_kernel performs inference for a single input vector. Each thread computes the dot product between that input and one support vector. The result of each thread is stored in a shared memory array and reduced using a tree-based parallel sum. The final result is written back to global memory using atomicAdd, as multiple blocks may contribute to the same output location. This version is useful for correctness validation and profiling but is not scalable to large batches.

In the BLOCK mode, implemented by svm_inference_batched_kernel, each block is assigned to one input vector. Threads within the block each handle a different support vector, computing their contribution to the decision function. As in the SINGLE kernel, partial results are written to shared memory and reduced cooperatively. The final result is written by thread 0 in each block. This model scales well with moderate batch sizes and is ideal when the number of support vectors is not too large to exceed shared memory limits. The use of dynamically allocated shared memory allows partial sum buffers to adapt to different block sizes.

In the THREAD mode, svm_inference_thread_parallel_kernel is designed for maximum scalability. Each thread independently processes one input vector, looping over all support vectors to compute the total decision value. To avoid redundant global memory reads, all support vectors are

cooperatively loaded into shared memory at the beginning of each block's execution. Threads then reuse this cached data while accessing their respective input vectors from global memory. This approach achieves high throughput when the number of input vectors is large and the support vector matrix fits in shared memory.

The following are the adjustable parameters:

- SVM_DIM: Specifies the dimensionality of input and support vectors. This directly affects dot product length, shared memory requirements, and loop bounds.
- SVM_NUM_SV: Determines the number of support vectors. Larger values increase computation per input and shared memory demand.
- NUM_TEST: Controls the total number of input vectors used during inference. Affects overall GPU workload and the grid size in batched kernel launches.
- BLOCK_SIZE: Sets the number of threads per block for the SINGLE and BLOCK kernels. Affects shared memory size, reduction depth, and parallelism granularity.
- THREAD_BLOCK_SIZE: Used specifically in THREAD mode, determining how many threads operate concurrently per block. Larger values improve warp utilization, but may increase register pressure or shared memory contention.

C. FPGA Implementation

All versions of our FPGA SVM inference accelerator used 2s complement representation, fixed-point precision, input and output M9Ks, and 18-bit multipliers. The designs were written in SystemVerilog RTL in VS Code and Quartus. Quartus was used for synthesis, simulation, P&R, timing analysis, and compilation onto our FPGA. Simulations were run in QuestaSim, and actual hardware tests were performed on our FPGA to prove functionality.

The initial version of our design aimed to achieve ultra-low latency, and used minimal pipelining as a result (in order to remove register latency). The design focused on maximum parallelism, specifically targeting the 30 testing data points by storing the input features in 60 M9K ROMs. These values were in 18-bit precision (6-bit integer, 12-bit fraction) in order to maximize but not exceed the multiplier unit capabilities. The design (Figure 6, 7) used 60 18-bit multipliers (2 per sample), and stored outputs in one M9K unit. This design's correctness was verified both in simulation (Figure 8), and was successfully run on actual hardware at the FPGA's native 50MHz clock.

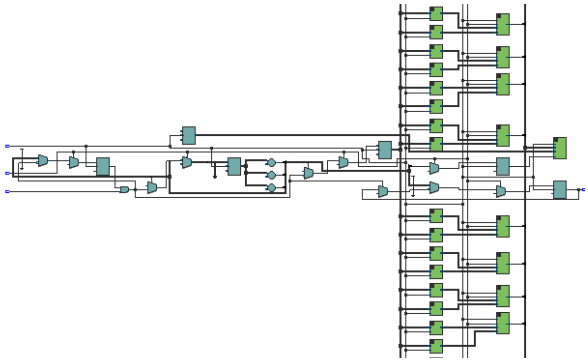

Figure 6: Synthesized design: FSM controlling address and control signals on the left, M9Ks and multipliers plus supporting logic on the right. Image has been trimmed for clarity.

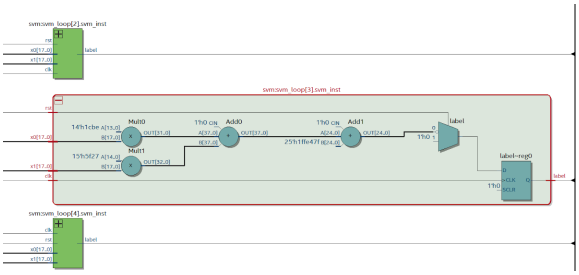

Figure 7: Synthesized SVM core: multipliers, adders, MUX, and register.

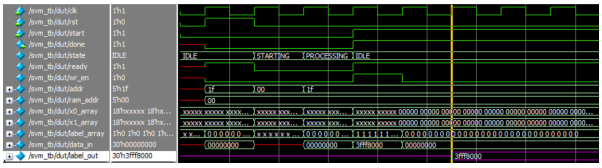

Figure 8: Simulation data showcasing 100% test accuracy. Output 0x3fff8000 represents 15 1's and 15 0's, properly inferring the correct classes.

Though the design was run at 50MHz, the synthesized design should be able to run at at least 540MHz even at the slow-slow corner (Figure 9).

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	539.96 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 9: Timing results for the slow-slow process corner.

However, the only way to achieve a higher-than-native clock frequency on the FPGA was to use a PLL, so we used the Quartus IP to instantiate a PLL and output a new 250MHz clock. This clock frequency was chosen as the restricted max clock frequency of the board as shown in Figure 9. While the PLL and the design could be run faster, we wired the top level design to start based on FPGA board IO, placing an upper limit of 250MHz on the actual clock frequency. Regardless, the design was recompiled with the PLL at 250MHz, and produced a design that could be theoretically clocked even faster, at about 583MHz (Figure 10).

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	583.09 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 10: Timing results for the slow-slow process corner with a PLL.

However, to actually verify the design was working at this high frequency on the FPGA, we needed to either check the output M9K memory after running the design, or add an indicator with self-checking logic. We opted for the latter just to make checking much easier, despite the unrealistic extra logic, just for proof of concept. Logic was added to turn on an LED if the M9K output was exactly equal to 0x3fff8000 (Figure 11). This design's extra logic drastically reduced the operating frequency down to 218MHz at the slow-slow corner (Figure 12), but ended up working at 250MHz on the FPGA (Figure 13, 14). This made logical sense as the slow-slow corner is overly pessimistic, and the design was running cold.

```
//self checking + sticky bit
done <= (data_in == 30'h3fff8000) ? 1'b1 : 1'b0 | done;
```

Figure 11: Self-checking logic verifying the data in the output M9K is correct. The done signal is routed to one of the FPGA board's LEDs.

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	218.77 MHz	218.77 MHz	clk	

Figure 12: Timing results for the slow-slow process corner with a PLL and self-checking logic.

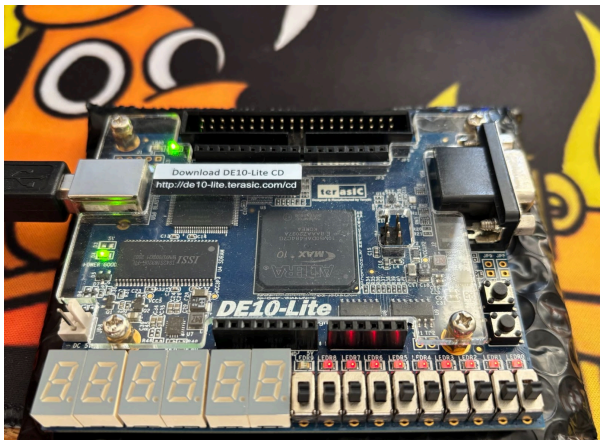

Figure 13: FPGA on startup: LED9 off, buttons are used to reset (initialize) and start the FSM.

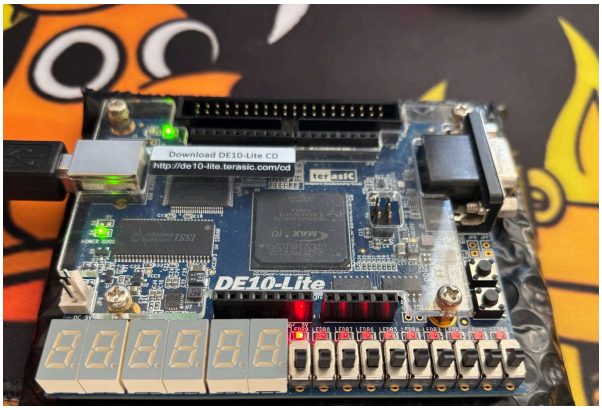

Figure 14: FPGA after resetting and starting sets the LED9 to ON, indicating the sticky bit in the self-checking logic triggered and passed, latching the LED high.

After verifying successful operation at 250MHz, all further iterations of the design tested on actual hardware removed the self-checking logic and reverted to just lighting the LED when complete. The timing analyzer was used to identify the max potential clock frequency, despite our hardware only safely running at 250MHz.

Two main changes were made in order to further bolster the performance (which already exceeded the capabilities of the board): low precision and pipelining.

After testing on the Python SVM model, a minimal fixed-point precision scheme of 7-bits (5-bit integer, 2-bit fraction) was found to still have perfect accuracy on the test data. Helper functions in Python were used alongside shell scripts to help update the weights and input values to this new precision in our RTL. The bit width parameters were adjusted, and the new model was compiled. However, to our surprise the low-precision model still had the maximum frequency limit from before (Figure 15).

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	583.09 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 15: Timing results for the slow-slow process corner with reduced precision.

This was most likely due to the critical path not being affected by bit precision, which sounds strange at first. However, we are using M9K memory units and 18-bit multipliers, which may have constant performance and access times regardless of utilization. In addition, the FPGA fabric has a fixed bit-width internally, so if 18-bits was not fully utilizing the interconnects, changing to 7-bits may not change the number of interconnects or change the routing (and therefore the delays in the critical path). Similarly, it is very possible that the compiler inferred M9K units for the weights instead of LUTs, so changing the bit-precision would still use M9K units and not reduce the number of LUTs.

To further test the critical path theory, and to experiment with a high-throughput design, a pipelined version of the design was implemented. The basic structure remained the same, although with two key changes. The SVM compute core was fully pipelined, inserting pipeline registers between the multiply

stage, add stage, and classification (bit select) stage. However, to account for this increased latency, the FSM in the top level needed two additional stages, requiring more LUTs. The resulting timing analysis ironically showed the pipelined design having a lower max frequency (Figure 16).

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	580.38 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 16: Timing results for the slow-slow process corner with pipelining.

This reduced frequency seems counterintuitive initially, but this reveals that the critical path is not in the SVM computation core. Rather, the increased delay alongside the increased logic in the top level FSM means that the critical path is most likely in the FSM itself. However, pipelining the FSM is not possible since each section is already pipelined, and all unnecessary signals were already removed. As a result, the design cannot be further improved latency-wise, unless an algorithmic or architectural change was made to the design.

V. EVALUATION

A. CPU & GPU Analysis (SVM1)

BLOCK_PARALLEL_DIM (svm1)

DIM	kernel_time_us	achieved_occupancy	IPC	flop_count_sp	Speedup
2	17,958	0.086	0.42	120,000,000	30.47x
8	16,167	0.084	0.42	240,000,000	87.33x
32	17,902	0.233	0.68	720,000,000	234.94x

As the input dimension increases from 2 to 32, the number of floating-point operations required by the GPU kernel grows linearly. However, the kernel time remains roughly constant across all three configurations, hovering around 17,000 µs. This indicates that the GPU is able to efficiently absorb the increased workload without a corresponding rise in latency. Meanwhile, CPU time increases significantly with dimension, which drives up the observed speedup from 30× to 234×. The IPC and occupancy also improve slightly at DIM=32, confirming better core utilization as the kernel performs more computations per thread.

THREAD_PARALLEL_DIM

DIM	kernel_time_us	occupancy	IPC	dram_read (GB/s)	flop_count	Speedup
2	566	0.077	0.13	132	40M	933.2x
8	614	0.475	0.4	484	170M	2300x
32	4,573	0.496	0.12	502	650M	907x

The thread-parallel kernel performs best at DIM=8, reaching a peak speedup of 2299×. This is the result of high memory throughput (483 GB/s), reasonable occupancy (47%), and much more computation per input. At DIM=2, low IPC and occupancy indicate underutilized hardware, though performance is still decent due to small kernel time. DIM=32 reduces speedup due to a large jump in kernel time to 4572 µs, despite high DRAM throughput and similar occupancy. This suggests pressure on shared memory, register usage, or instruction-level parallelism limitations with deeper loop unrolling.

THREAD_PARALLEL_NUM_INPUTS

Inputs	kernel_time_us	dram_util	flop_count	Speedup
30	2.78	1	120	0.72x
1,000	2.62	0	4,000	19.0x
100,000	8.48	1	400,000	700.1x
10,000,000	565.47	4	40M	940.1x

This table highlights how the GPU is particularly efficient when processing large batches of inputs. For 30 inputs, the kernel time is only 2.78 µs, which means kernel launch and memory copy overheads dominate — resulting in sub-1 speedup. Once the batch grows to 100K or 10M inputs, the kernel scales exceptionally well, maintaining low latency relative to the large number of operations. The dramatic increase in flop count and corresponding increase in DRAM utilization further support the claim that the GPU becomes more effective with larger batches.

THREAD_PARALLEL_THREADS

Threads Max	kernel_time_us	occupancy	IPC	Speedup
4	4,043	0.055	0.12	132.6x
8	2,023	0.058	0.11	261.8x
32	565	0.077	0.13	973.0x
64	284	0.152	0.22	1866.5x

Increasing the number of threads per block directly improves performance. As THREADS_MAX increases from 4 to 64, kernel time drops by over an order of magnitude, while both occupancy and IPC increase. This reflects greater warp-level parallelism and improved utilization of the GPU's streaming multiprocessors. The higher speedups observed at 32 and 64 threads demonstrate that the kernel is highly scalable with respect to thread count in this configuration.

B. CPU & GPU Analysis (SVM2)

THREAD_SVM_NUM_SV

SV count	kernel_time_us	occupancy	IPC	dram_read	flop_count	Speedup
16	615.3	0.958	3.15	122.37	1.13B	9725.3x
32	1197.1	0.969	3.13	62.77	2.25B	9578.9x

The thread-parallel kernel for the dual SVM form delivers extremely high speedups, achieving nearly 10,000× acceleration over the CPU. With 16 support vectors, kernel time is only ~615 µs despite computing over a billion FLOPs. When the number of support vectors doubles to 32, kernel time increases linearly, which is expected due to the doubling of computation. However, DRAM throughput drops significantly, implying possible memory contention or shared memory limits. The IPC remains very high (over 3.1), indicating excellent instruction-level utilization even at high workloads.

BLOCK_SVM_NUM_SV

SV count	kernel_time_us	occupancy	IPC	dram_read	flop_count	Speedup
16	34166	0.894	3.03	2.6	3.52B	175.1x
32	30701	0.893	3.03	2.61	4.48B	374.0x

In the block-parallel kernel for dual SVM, speedup improves from 175× to 374× when the number of support vectors is doubled. Interestingly, kernel time drops slightly, which may seem counterintuitive. This is likely due to better GPU scheduling and reduced idle cycles as the computation per block increases. The IPC and occupancy are high and stable, and DRAM throughput remains roughly constant, which

confirms that the GPU efficiently handles increased compute without becoming memory-bound in this mode.

THREAD_THREAD_BLOCK_SIZE

Threads	kernel_time_us	occupancy	IPC	Speedup
32	685.6	0.482	3.01	8723.8x
64	618.9	0.957	3.18	9665.8x
128	611.1	0.958	3.16	9788.9x
256	616	0.954	3.13	9704.4x

The thread-parallel kernel with varying block size demonstrates that performance saturates at block sizes of 64 or more. As block size increases, the achieved occupancy and IPC rise, indicating better warp-level execution and pipeline usage. Speedup improves rapidly from 8723× at block size 32 to nearly 9800× at 128, then flattens — showing that the kernel reaches optimal occupancy and instruction throughput. The consistent kernel times above 64 threads also suggest that shared memory and register usage are no longer limiting factors past that point.

C. FPGA Analysis

The FPGA timing results indicate that the optimal design is the non-pipelined design using a PLL for higher frequency, and using 18-bit precision. This design optimizes both throughput and latency, and cannot be further pipelined to improve performance. The design achieves 100% test accuracy for the Iris test data, and theoretically can run on actual hardware at at least 583.09MHz. Since all data can be processed (and classified) in one cycle, the latency is 1.715ns. Accounting for the memory writes for input and output, there is an additional one cycle each. As a result, the total inference time for the Iris testing data is 1.715ns*3 or 5.145ns. The design therefore has a throughput of 30/5.145ns, as it outputs 30 samples every 1.715ns, though this can be scaled up. If the pipeline is fully utilized, the design's throughput would be 30/1.715ns.

If a larger dataset were to be run, the design could also be scaled to use more M9Ks and multipliers, while keeping the non-pipelined, low latency design since it is faster than the fully pipelined design. This would have a latency of about CEILING(N/U)*1.715ns, where N is the number of total samples, and U is the number of samples the FPGA is processing at once. U is estimated to be 72, as each data point needs two multipliers and there are 144 multipliers on the chip, though 100% utilization will likely cause degradations in the routing and make the latency a bit longer as a result.

It is worth noting that non-linear and multi-class SVMs were not implemented on the FPGA, and we rather focused on linear, binary classification in order to keep the primary goal focused: optimize the FPGA hardware to the absolute limit for a given inference task. Had we implemented the non-linear or multiclass SVM inference core on the FPGA, the design process would be exactly the same: identify our resources available, plan out which resources to use when, and iteratively improve on the design to minimize latency.

D. Comparisons

Various tests were run on the GPU to explore the opportunities to accelerate SVM inference cores, but its performance for the Iris dataset is of particular interest when compared to the FPGA implementation. As a reference, the CPU SVM inference kernel execution time was 2us. The GPU kernel execution time is composed of three separate phases: CPU overhead, CUDA memcpy, and the actual GPU kernel computation. The actual kernel execution time had a latency of 2.78us. But including the necessary CPU and memcpy overheads, the total execution time took 133ms total. The latencies of the three devices are compared at the log scale in Figure 17.

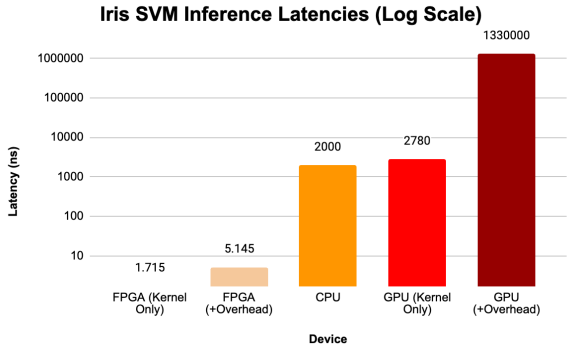

Figure 17: SVM inference latencies across devices for the Iris test dataset.

This experimental data showcases the effectiveness of FPGAs when operating on small datasets, capable of achieving ultra-low latencies by running the bare minimum computations possible with very little overhead. However, though the GPU has a very high overhead due to CPU setup times and the memcpy times, these are one or two time costs that can be amortized over time for very large datasets. FPGAs struggle to compete with GPUs on the throughput side due to hardware constraints, highlighting the importance of using the right device for the right application and dataset. GPUs excel at computing lots of data in parallel over large amounts of time, while FPGAs seem to excel at processing smaller datasets very quickly with minimal overhead.

VI. CONCLUSIONS

On the FPGA side, our architecture exploration made it clear that even small FPGAs, when fully utilized and optimized, can achieve very low latencies (single digit nanoseconds) for SVM inference, though they are limited by their size (max throughput of 72 data points per cycle for our FPGA). However, scaling the same principles used to optimize this design to bigger FPGAs reinforces prior work, proving that FPGAs can excel at low latency inference with decently high throughput, limited mostly by the FPGA's size and hardware resources available. Another major limitation is the dataset size, and it becomes clear from this work that performing SVM at the edge would be difficult when working with very large amounts of data due to throughput limitations. In such cases, GPUs would be better suited for such tasks. Or even more so, running DNNs on GPUs to more efficiently and accurately classify large amounts of data. But for edge devices and small datasets, FPGAs can perform quite well when the design is optimized and tailored for the hardware dimensions and capabilities. Even small FPGAs can outperform expensive

GPUs as seen with the Iris dataset, and our results highlight the importance of using the correct device for the correct application and dataset; in this case, FPGAs are ideal for SVM inference which typically deal with smaller datasets, due to their minimal overhead. GPUs on the other hand, excel at processing very large datasets in parallel with very high throughput, making them ideal for larger, heavier inference tasks than those typically processed with SVMs.

VII. STATEMENT OF WORK

Below we discuss how the work was primarily split. Although we spearheaded the work/portions discussed below for each person, we made sure to offer help and be involved with the design decisions on both sides.

Taylor worked on the initial Python SVM model's training and testing. He worked on the full FPGA side of the project, conducting the FPGA architecture research, design space exploration, full RTL design flow, and FPGA testing and analysis.
Arunan worked on expanding the initial Python SVM model to a more complex version that also utilized support vectors, alphas, and labels. Then, he worked on the full CPU benchmark and GPU kernel implementations of the SVM, both for SVM (using weights without alphas, SV's, etc.) and for SVM (using alphas, SV's, etc.). To present a broader applicability to the project, he also ensured that the code was expandable to larger DIM values. Finally, he wrote the Makefile and Python scripts that ran all the various tests and parsed the output data, and did analysis on these comparisons.

REFERENCES

[1] Sasidharan, Aswathi. "Support Vector Machine Algorithm." *GeeksforGeeks*, 20 Jan. 2021, www.geeksforgeeks.org/support-vector-machine-algorithm/.

[2] Wang, Pin, et al. "Comparative Analysis of Image Classification Algorithms Based on Traditional Machine Learning and Deep Learning." *Pattern Recognition Letters*, vol. 141, Aug. 2020, pp. 61–67, https://doi.org/10.1016/j.patrec.2020.07.042.

[3] Lee, Sarah. "SVM vs Deep Learning: 6 Critical Differences Impacting ML Outcomes." *Numberanalytics.com*, 2025, www.numberanalytics.com/blog/svm-vs-deep-learning-6-differences? Accessed 13 June 2025.

[4] Afifi, Shereen, et al. "FPGA Implementations of SVM Classifiers: A Review." *SN Computer Science*, vol. 1, no. 3, 23 Apr. 2020, https://doi.org/10.1007/s42979-020-00128-9.

[5] "The Iris Dataset." *Scikit-Learn*, 2024, scikit-learn.org/1.4/auto_examples/datasets/plot_iris_dataset.html. Accessed 12 June 2025.

[6] scikit-learn. "Sklearn.svm.SVC — Scikit-Learn 0.22 Documentation." *Scikit-Learn.org*, 2019, scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. Accessed 12 June 2025.

[7] *ALTERA M9K EMBEDDED MEMORY BLOCKS.* https://faculty-web.msoe.edu/johnsontimoj/Common/FILES/DE10_Lite_User_Manual.pdf

[8] Wikipedia Contributors. "CORDIC." *Wikipedia*, Wikimedia Foundation, 8 July 2019, en.wikipedia.org/wiki/CORDIC. Accessed 12 June 2025.

[9] *DE10-Lite User Manual 1*. 2017. https://www.ece.ucdavis.edu/~bbaas/180/notes/Handout.M9K.mems.pdf

[10] *3.2.1. Shift-And-Add Multiplication*.

[11] Catanzaro, Bryan & Sundaram, Narayanan & Keutzer, Kurt. (2008). Fast support vector machine training and classification on graphics processors. Proceedings of the 25th International Conference on Machine Learning. 104-111. 10.1145/1390156.1390170.

[12] "Welcome to CUML's Documentation!#." Welcome to cuML's Documentation! - Cuml 25.06.00 Documentation, docs.rapids.ai/api/cuml/stable/. Accessed 13 June 2025.

Appendix

Showcasing speedups (CPU/GPU times):

Speedups from: parsed_outputs1.csv

[BLOCK_PARALLEL_DIM]
BLOCK_PARALLEL_DIM_2.txt: Speedup = 30.47x
BLOCK_PARALLEL_DIM_8.txt: Speedup = 87.33x
BLOCK_PARALLEL_DIM_32.txt: Speedup = 234.94x

[BLOCK_PARALLEL_NUM]
BLOCK_PARALLEL_NUM_INPUTS_30.txt: Speedup = 0.83x
BLOCK_PARALLEL_NUM_INPUTS_1000.txt: Speedup = 12.40x
BLOCK_PARALLEL_NUM_INPUTS_100000.txt: Speedup = 32.16x
BLOCK_PARALLEL_NUM_INPUTS_10000000.txt: Speedup = 29.95x

[BLOCK_PARALLEL_THREADS]
BLOCK_PARALLEL_THREADS_MAX_4.txt: Speedup = 33.38x
BLOCK_PARALLEL_THREADS_MAX_8.txt: Speedup = 29.53x
BLOCK_PARALLEL_THREADS_MAX_32.txt: Speedup = 33.14x
BLOCK_PARALLEL_THREADS_MAX_64.txt: Speedup = 33.73x

[THREAD_PARALLEL_DIM]
THREAD_PARALLEL_DIM_2.txt: Speedup = 933.21x
THREAD_PARALLEL_DIM_8.txt: Speedup = 2299.96x
THREAD_PARALLEL_DIM_32.txt: Speedup = 906.99x

[THREAD_PARALLEL_NUM]
THREAD_PARALLEL_NUM_INPUTS_30.txt: Speedup = 0.72x
THREAD_PARALLEL_NUM_INPUTS_1000.txt: Speedup = 19.05x
THREAD_PARALLEL_NUM_INPUTS_100000.txt: Speedup = 700.12x
THREAD_PARALLEL_NUM_INPUTS_10000000.txt: Speedup = 940.13x

[THREAD_PARALLEL_THREADS]
THREAD_PARALLEL_THREADS_MAX_4.txt: Speedup = 132.61x
THREAD_PARALLEL_THREADS_MAX_8.txt: Speedup = 261.80x
THREAD_PARALLEL_THREADS_MAX_32.txt: Speedup = 973.02x
THREAD_PARALLEL_THREADS_MAX_64.txt: Speedup = 1866.52x

Speedups from: parsed_outputs2.csv

[svm2_BLOCK_BLOCK]
svm2_BLOCK_BLOCK_SIZE_32.txt: Speedup = 334.10x
svm2_BLOCK_BLOCK_SIZE_64.txt: Speedup = 372.32x
svm2_BLOCK_BLOCK_SIZE_128.txt: Speedup = 372.95x
svm2_BLOCK_BLOCK_SIZE_256.txt: Speedup = 194.56x

[svm2_BLOCK_SVM]
svm2_BLOCK_SVM_NUM_SV_16.txt: Speedup = 175.08x
svm2_BLOCK_SVM_NUM_SV_32.txt: Speedup = 374.02x

[svm2_THREAD_SVM]
svm2_THREAD_SVM_NUM_SV_16.txt: Speedup = 9725.32x
svm2_THREAD_SVM_NUM_SV_32.txt: Speedup = 9578.90x

[svm2_THREAD_THREAD]
svm2_THREAD_THREAD_BLOCK_SIZE_32.txt: Speedup = 8723.83x
svm2_THREAD_THREAD_BLOCK_SIZE_64.txt: Speedup = 9665.78x
svm2_THREAD_THREAD_BLOCK_SIZE_128.txt: Speedup = 9788.87x
svm2_THREAD_THREAD_BLOCK_SIZE_256.txt: Speedup = 9704.43x