



A Reconfigurable FPGA-based ML Library for Kernel Methods

Yousef Alnaser^{1, 2}, Jan Langer¹, Anna Zuchna²

¹ Fraunhofer-Institut für Elektronische Nanosysteme ENAS, Technologie-campus 3, 09126 Chemnitz

² Technische Universität Chemnitz, Str. der Nationen 62, 09111 Chemnitz

Introduction

Kernel Matrix Complexity

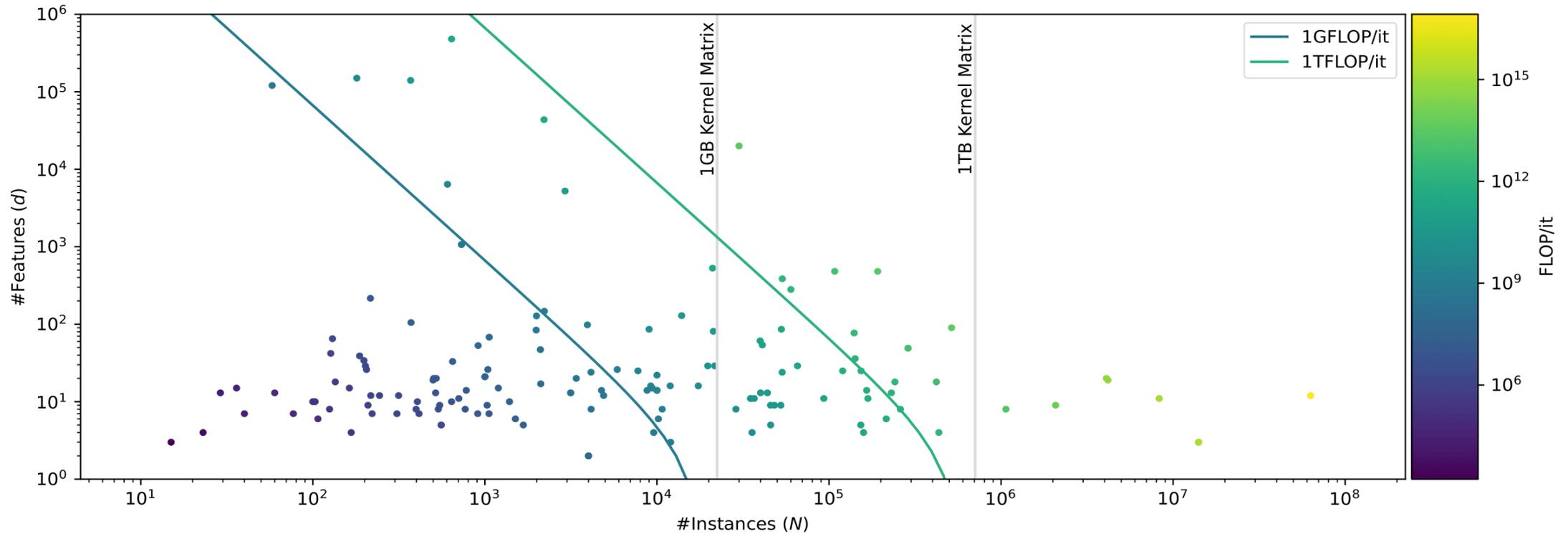
Scale poorly for large datasets

Radial Basis Kernel (RBF)

Input: $\mathcal{I} \in \mathbb{R}^{N \times d}$ Kernel: $K(\mathcal{I}, \mathcal{I}^T) = e^{-\gamma \|\mathcal{I} - \mathcal{I}^T\|^2} \in \mathbb{R}^{N \times N}$

Compute $\mathcal{O}(N^2d)$

Storage $\mathcal{O}(N^2)$



ML FPGA Library for Kernel Methods

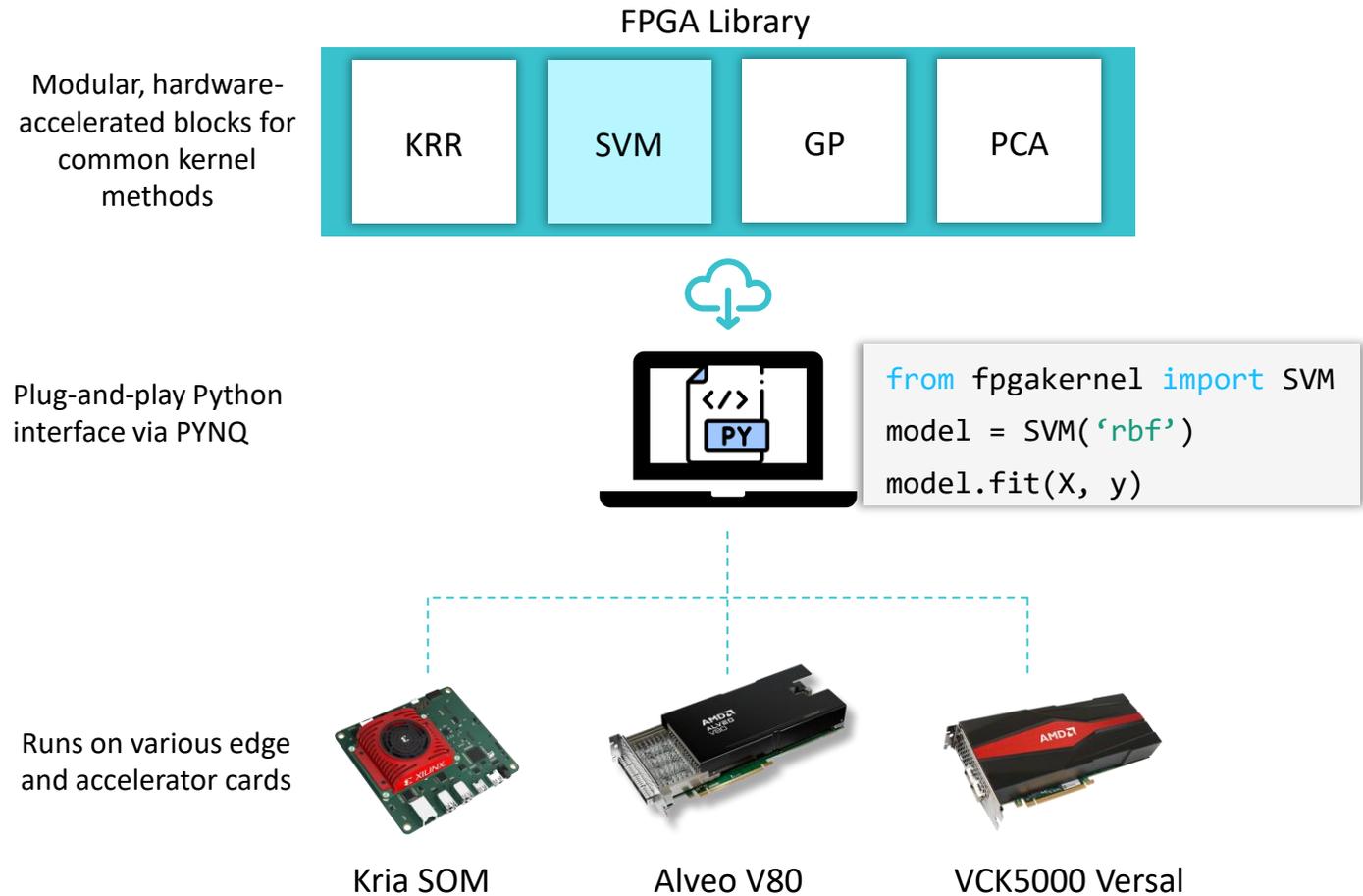
Hardware acceleration of kernel ML

Why this library?

Main Advantages

- Up to 30× faster than Python CPU libraries
- Partial reconfiguration
 - Support different kernels
 - Reduce Library Size
- Modular design: easy to extend with new solvers and kernels
- Works with big data, no FPGA knowledge needed

Accelerated kernel learning from Python — without hardware complexity.



Use Case

Kernel Ridge Regression

Formulation

$$\begin{aligned} b &= Ax && \in \mathbb{R}^N \\ A &= K(\mathcal{I}, \mathcal{I}^T) + \lambda I && \in \mathbb{R}^{N \times N} \\ \hat{x} &= cg(A, b, \epsilon) && \in \mathbb{R}^N \end{aligned}$$

Profiling

Complexities:

- Lines 1, 2, & 7 have Quadratic Complexity $O(N^2)$
 - Runs on FPGA fabric (HW)
- The remaining lines have Linear Complexity $O(N)$
 - Runs on ARM cores (SW)

Only operations with quadratic complexity are accelerated

Algorithm 1 KRR+CG

Require: x_0, ϵ, λ

- 1: $A = K(I, I^T) + \lambda \mathbb{1}_N$
- 2: $\mathbf{r}_0 := \mathbf{b} - A \cdot \mathbf{x}_0$
- 3: **if** $\|\mathbf{r}_0\| \leq \epsilon$ **then return** \mathbf{x}_0
- 4: $\mathbf{p}_0 := \mathbf{r}_0$
- 5: $k := 0$
- 6: **while** $\|\mathbf{r}_k\| \geq \epsilon$ **do**
- 7: $\mathbf{q}_k := A \cdot \mathbf{p}_k$
- 8: $\alpha_k := \frac{\mathbf{r}_k^T \cdot \mathbf{r}_k}{\mathbf{p}_k^T \cdot \mathbf{q}_k}$
- 9: $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
- 10: $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{q}_k$
- 11: $\beta_k := \frac{\mathbf{r}_{k+1}^T \cdot \mathbf{r}_{k+1}}{\mathbf{r}_k^T \cdot \mathbf{r}_k}$
- 12: $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
- 13: $k := k + 1$
- 14: **end while**

Introduction

HW/SW Partitioning

Operations of Interest:

- Kernel-Matrix (KM) Construction

$$A = K(\mathcal{I}, \mathcal{I}^T) + \lambda \mathbb{1}_N$$

- Matrix-Vector (MV) Multiplication

$$q = Ap$$

On-the-fly calculation → High memory BW requirements

Matrix Tiling/Partitioning → Reduced BW requirements

Algorithm 1 KRR+CG

Require: $x_0, \varepsilon, \lambda$

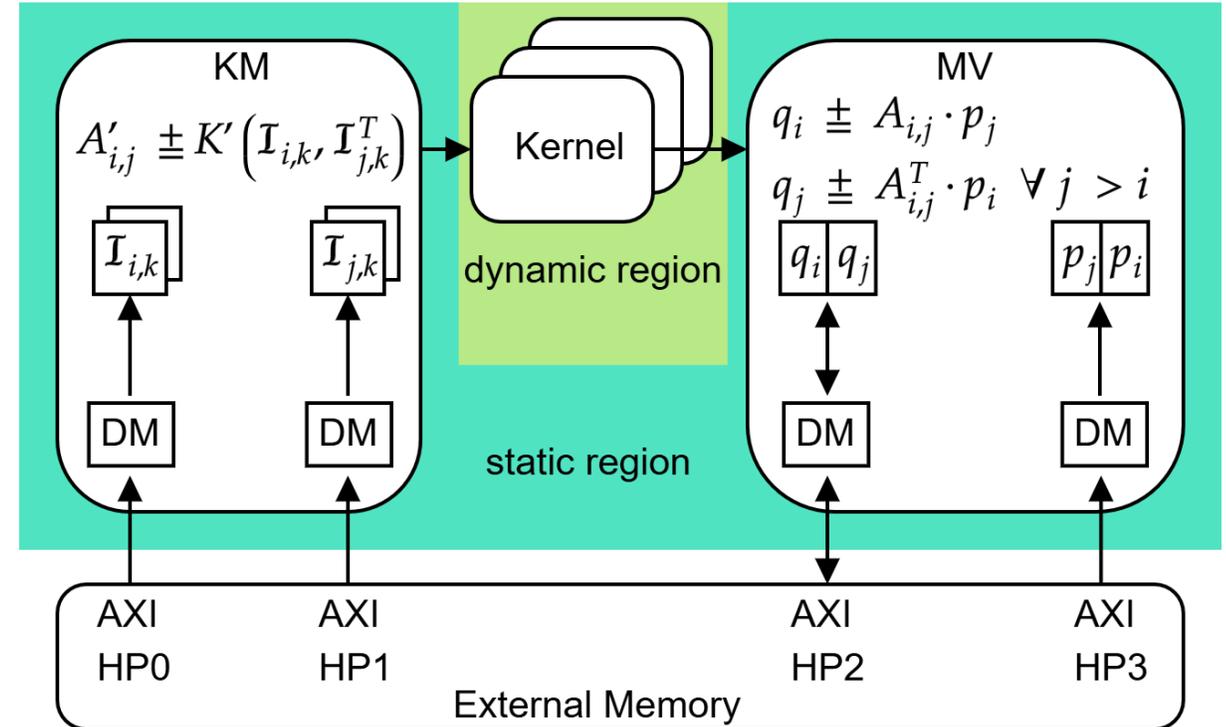
- 1: $\mathbf{r}_0 := \mathbf{b} - (K(\mathcal{I}, \mathcal{I}^T) + \lambda \mathbb{1}_N) \cdot \mathbf{x}_0$
 - 2: **if** $\|\mathbf{r}_0\| \leq \varepsilon$ **then** *return* \mathbf{x}_0
 - 3: $\mathbf{p}_0 := \mathbf{r}_0$
 - 4: $k := 0$
 - 5: **while** $\|\mathbf{r}_k\| \geq \varepsilon$ **do**
 - 6: $\mathbf{q}_k := (K(\mathcal{I}, \mathcal{I}^T) + \lambda \mathbb{1}_N) \cdot \mathbf{p}_k$
 - 7: $\alpha_k := \frac{\mathbf{r}_k^T \cdot \mathbf{r}_k}{\mathbf{p}_k^T \cdot \mathbf{q}_k}$
 - 8: $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
 - 9: $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{q}_k$
 - 10: $\beta_k := \frac{\mathbf{r}_{k+1}^T \cdot \mathbf{r}_{k+1}}{\mathbf{r}_k^T \cdot \mathbf{r}_k}$
 - 11: $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
 - 12: $k := k + 1$
 - 13: **end while**
-

Accelerator Implementation

Design Overview

Accelerator hardware

- Kernel Matrix (KM) Construction
- Reconfigurable Kernel Module
- Matrix-Vector (MV) Multiplication
- Implemented in HLS in 300 LoC
- Compile-time configurable
- Run-time compatible with any arbitrary dataset



Accelerator Implementation

Common Operations in Supported Kernels

Table 1 – Supported Kernels

Kernel Function	Equation
RBF	$e^{-\gamma \ I - I^T\ ^2}$
Laplacian	$e^{-\gamma \ I - I^T\ }$
Linear	$I \cdot I^T + c$
Polynomial	$(I \cdot I^T + c)^r$
Sigmoid	$\tanh(aI \cdot I^T + c)$

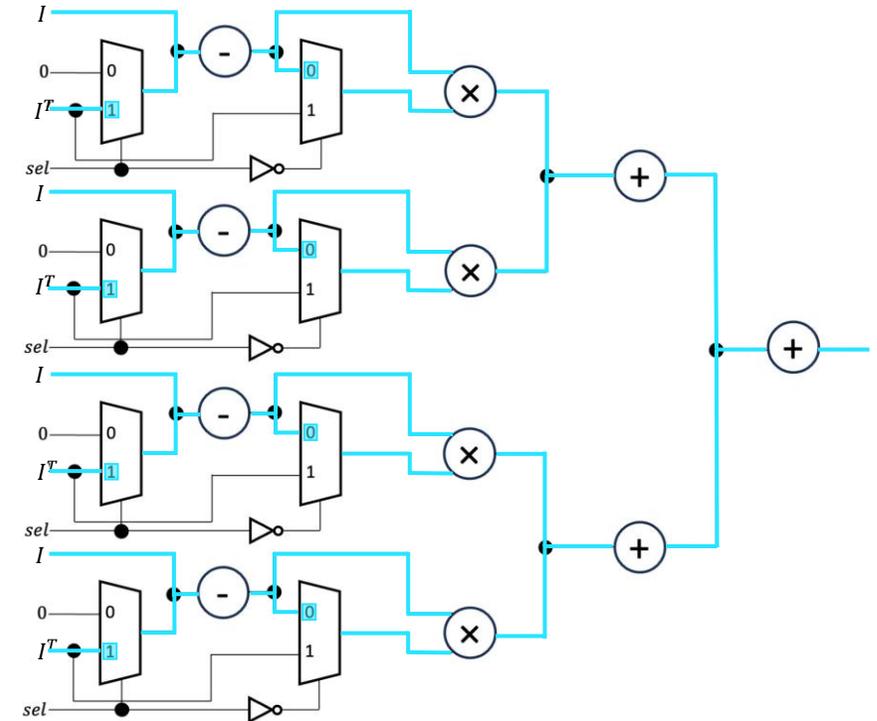


Fig. 4 – Kernel Implementation

Accelerator Implementation

Common Operations in Supported Kernels

Table 1 – Supported Kernels

Kernel Function	Equation
RBF	$e^{-\gamma \ I - I^T\ ^2}$
Laplacian	$e^{-\gamma \ I - I^T\ }$
Linear	$I \cdot I^T + c$
Polynomial	$(I \cdot I^T + c)^r$
Sigmoid	$\tanh(aI \cdot I^T + c)$

Common operations are combined in a single implementation

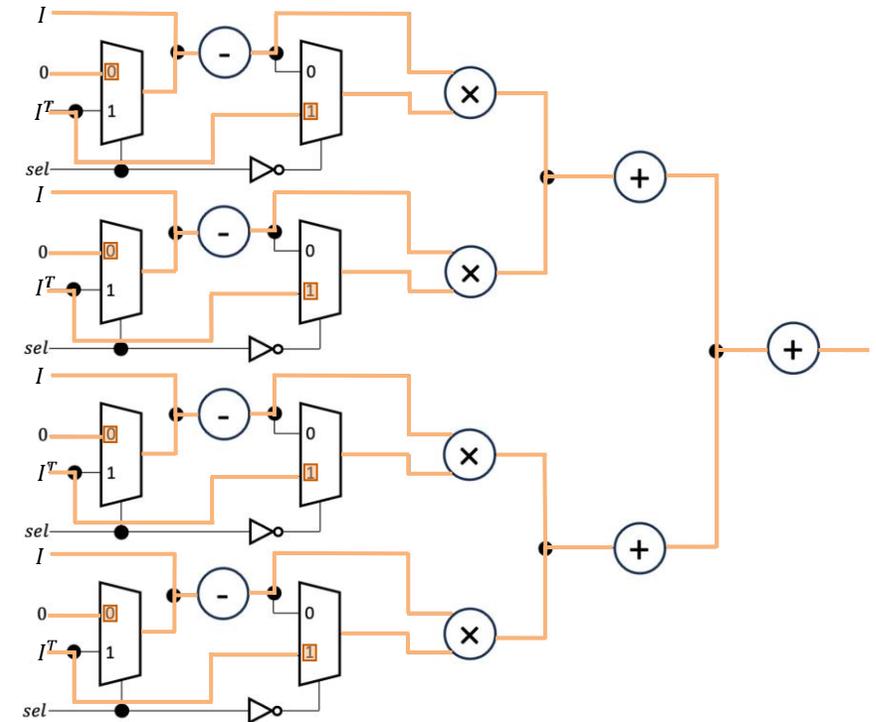


Fig. 4 – Kernel Implementation

Accelerator Implementation

Kernels Calculations

Common operations → kernel specific operations

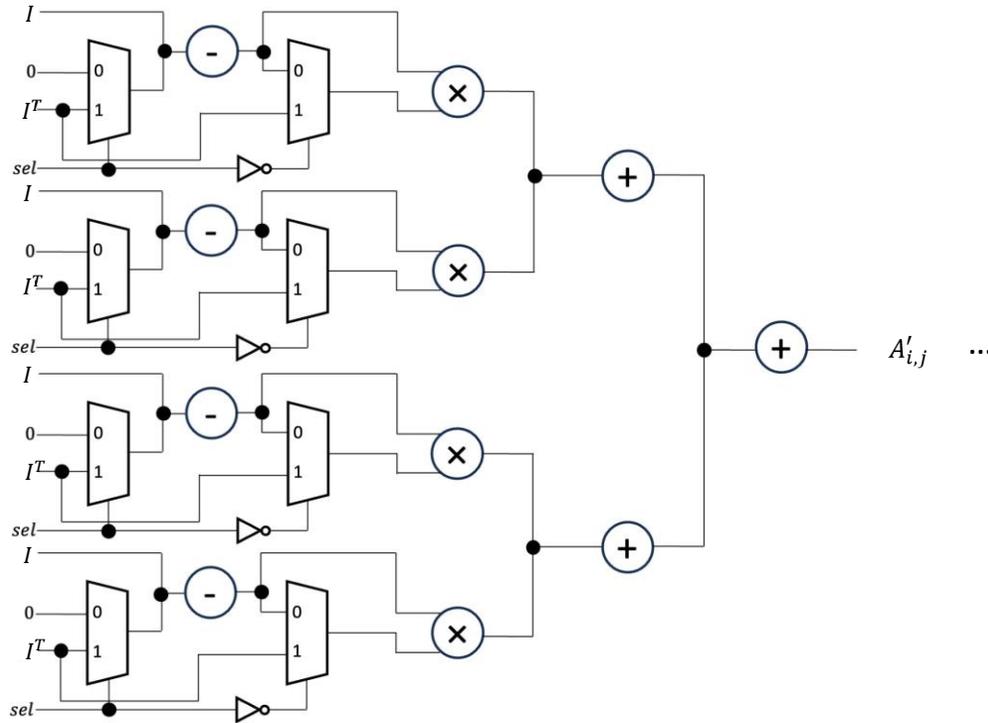


Fig. 4 – Kernel Implementation

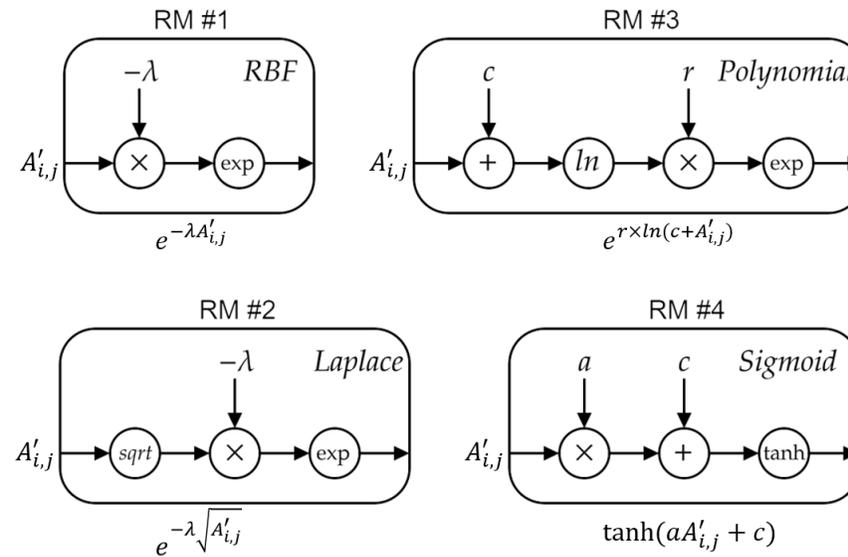


Fig. 5 – Reconfigurable modules for supported kernels

Accelerator Implementation

Matrix Partitioning

Macro-tiles

Kernel Matrix

- $A'_{i,j}$: macro-tile of size $T \times T$
 - $A'_{ij} = A'_{ji} \rightarrow$ only the upper triangle is computed and reused

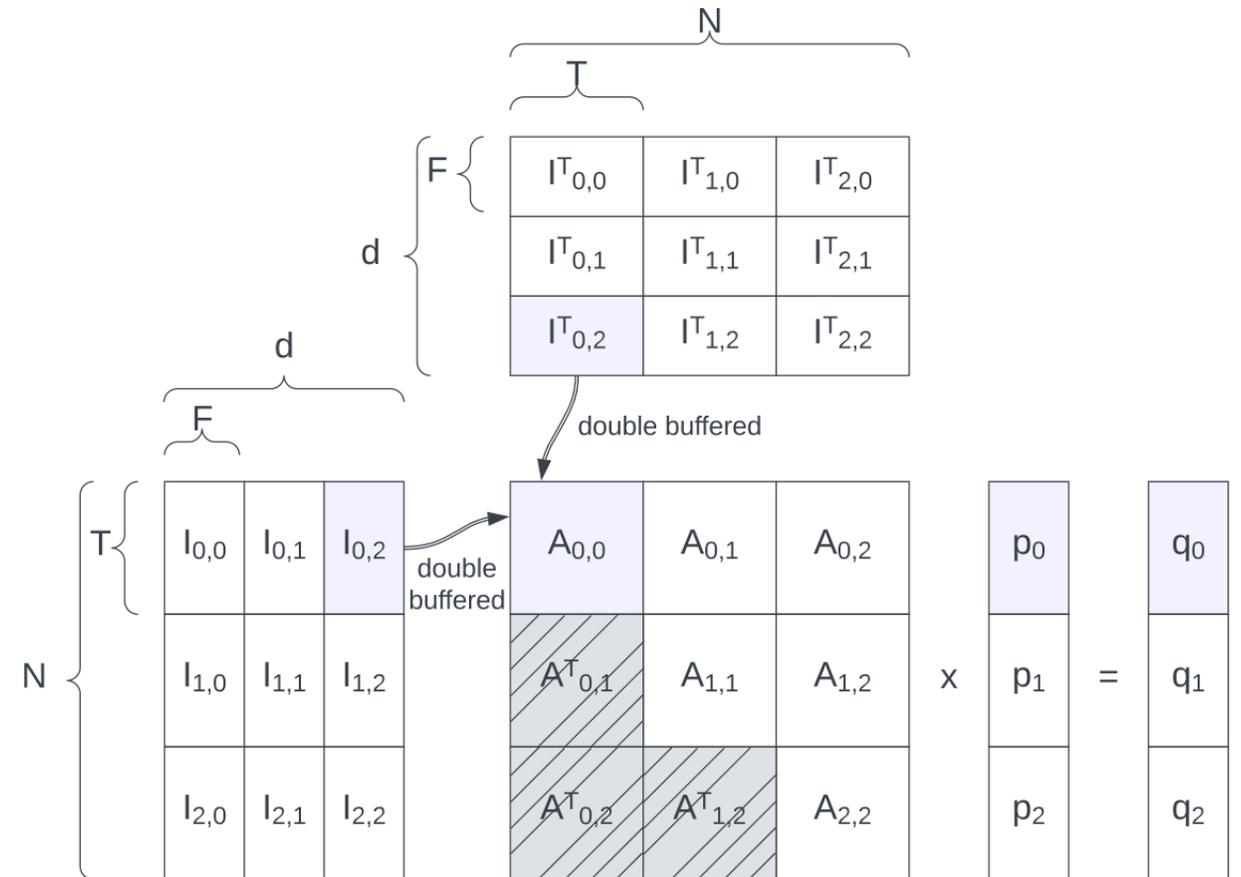
Input Matrix

- $I_{i,k}$: partitioned into $\frac{N}{T} \times \frac{d}{F}$ tiles each of size $T \times F$

MV multiplication

- Once one $A_{i,j}$ is calculated, both equation are computed:

Macro tiles for storage and BW



Results

Performance Evaluation

Comparison

- Python **Cholesky** (cubic complexity – not shown)
 - From `scikit-learn`
 - requires large memory storage
- Python **CG** – 30x slower for big data
 - Precomputes the kernel matrix
 - `scikit-learn`, `SciPy` and `NumPy`
 - requires large memory storage
- Python **Naïve** - 80x slower
 - Matrix partitioning to improve caching
 - `scikit-learn` and `NumPy`



Kria SOM

30x faster

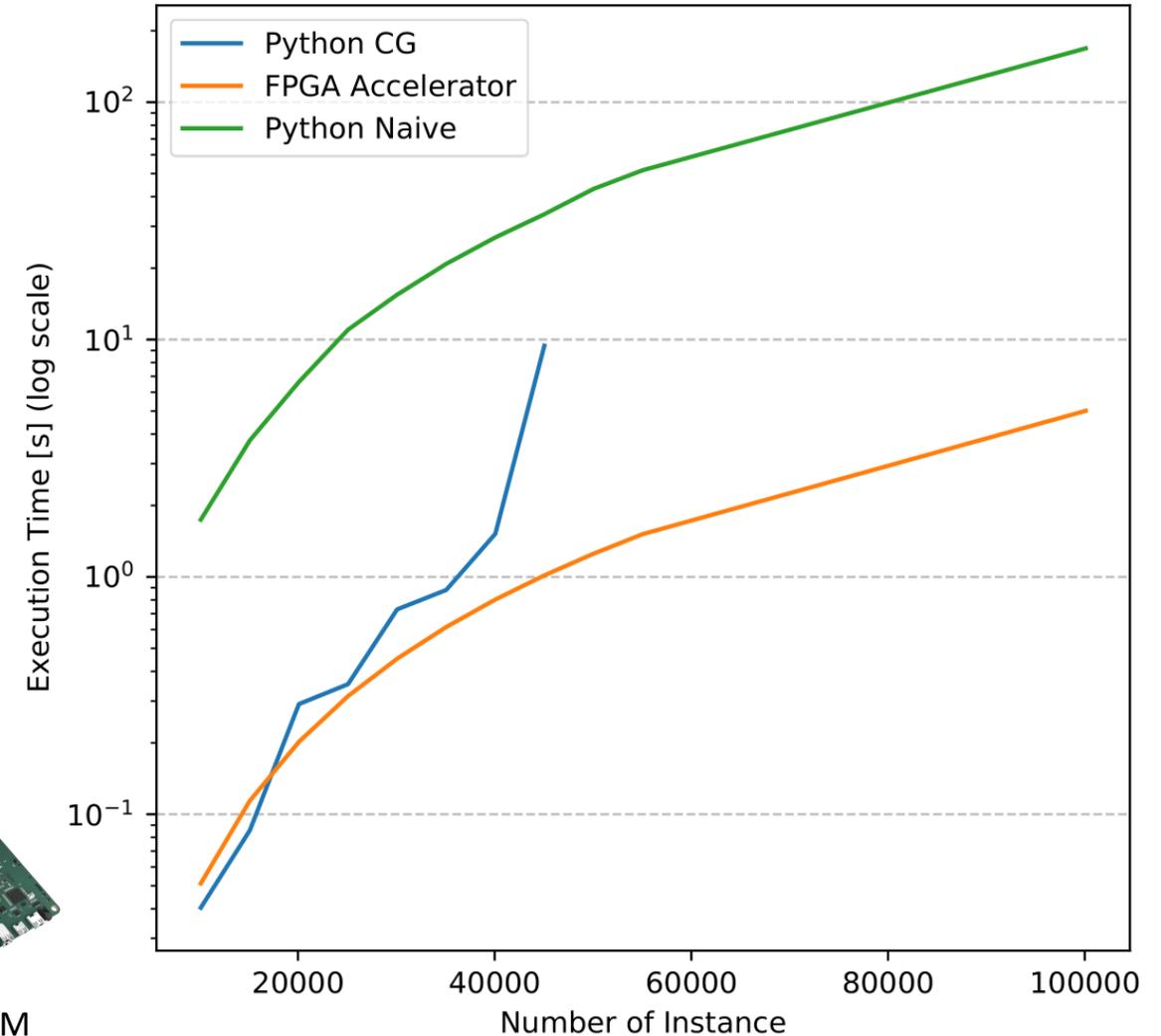


TABLE II: Resource Utilisation Comparison between an accelerator with all-static and an accelerator with PR implementations for different kernels.

Resource	Compile-time parameters ($T = 128, F = 10, P_x = 8, P_y = 1$)								
	Accelerator with all-static implementation				Accelerator with PR implementation				
	Static Region				Static Region	Dynamic Region (Allocated DSPs = 32.4%, LUT = 30.3%, FF = 31.0%, BRAM = 15.3%)			
	RBF	Laplacian	Polynomial + Linear	Sigmoid		RBF	Laplacian	Polynomial + Linear	Sigmoid
DSP (%)	60.9%	60.9%	57.7%	70.5%	54.4%	8.3%	8.3%	16.7%	29.5%
LUT (%)	67.5%	67.5%	57.5%	82.1%	62.7%	8.6%	8.8%	12.5%	27.5%
FF (%)	41.0%	41.1%	33.0%	69.4%	38.3%	3.2%	3.4%	5.0%	28.2%
BRAM (%)	58.3%	58.3%	58.3%	70.5%	51.4%	0.0%	0.0%	0.0%	13.9%
Sizes (MB)	8.018	8.018	8.019	8.019	8.124	2.778	2.779	2.779	2.779

Conclusion

Challenges

Partial Reconfiguration

- Rigid Resource Allocation
 - Must accommodate the largest RM
- Manual Floorplanning
 - Must manually assign PR regions
- Toolchain Limitations
 - PR in Vivado is not well-documented and buggy

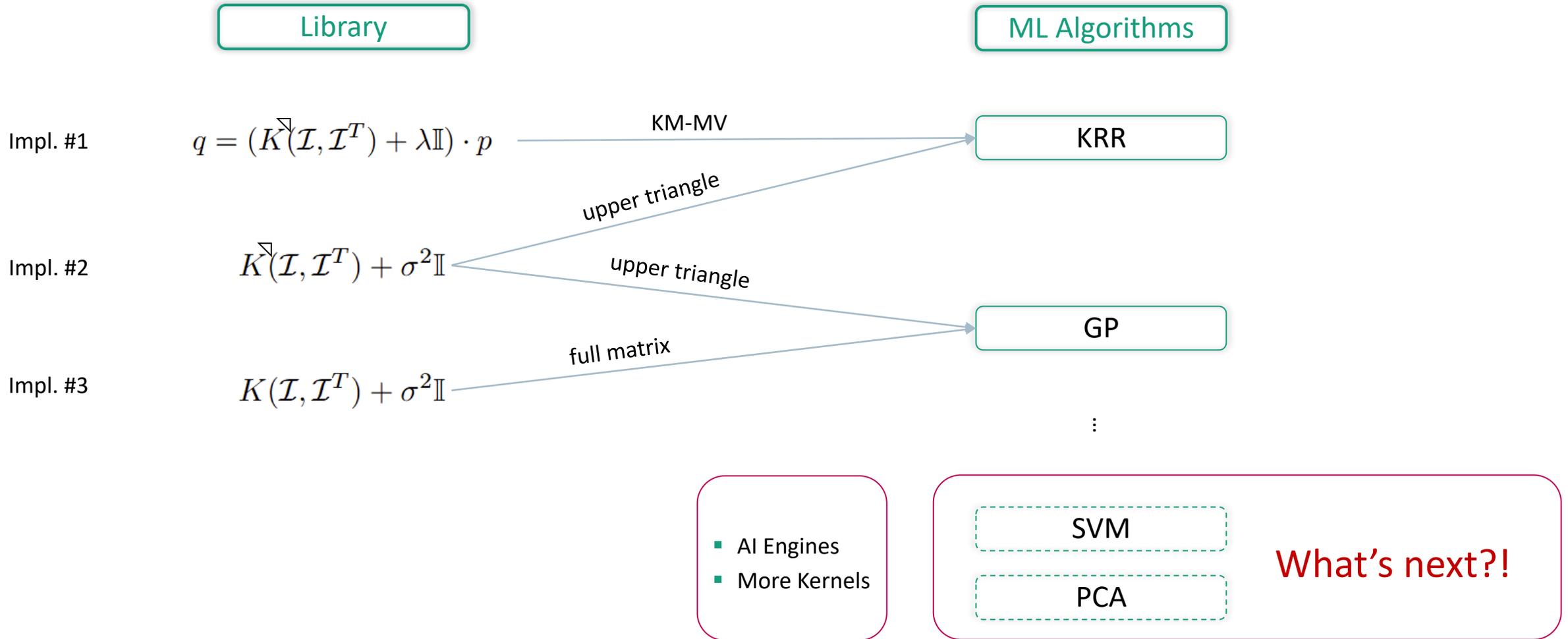
HLS

- Single-cycle accumulation of floating point
 - Must be manually handled.
- Loop dependencies
 - Predicts incorrect dependencies

Data Movement

- Manual Data handling
- Xilinx DMA LogiCore
 - Requires large amount of storage, for big data

Conclusion Summary





Thank you for your attention

Contact

Yousef Alnaser

yousef.alnaser@enas.fraunhofer.de