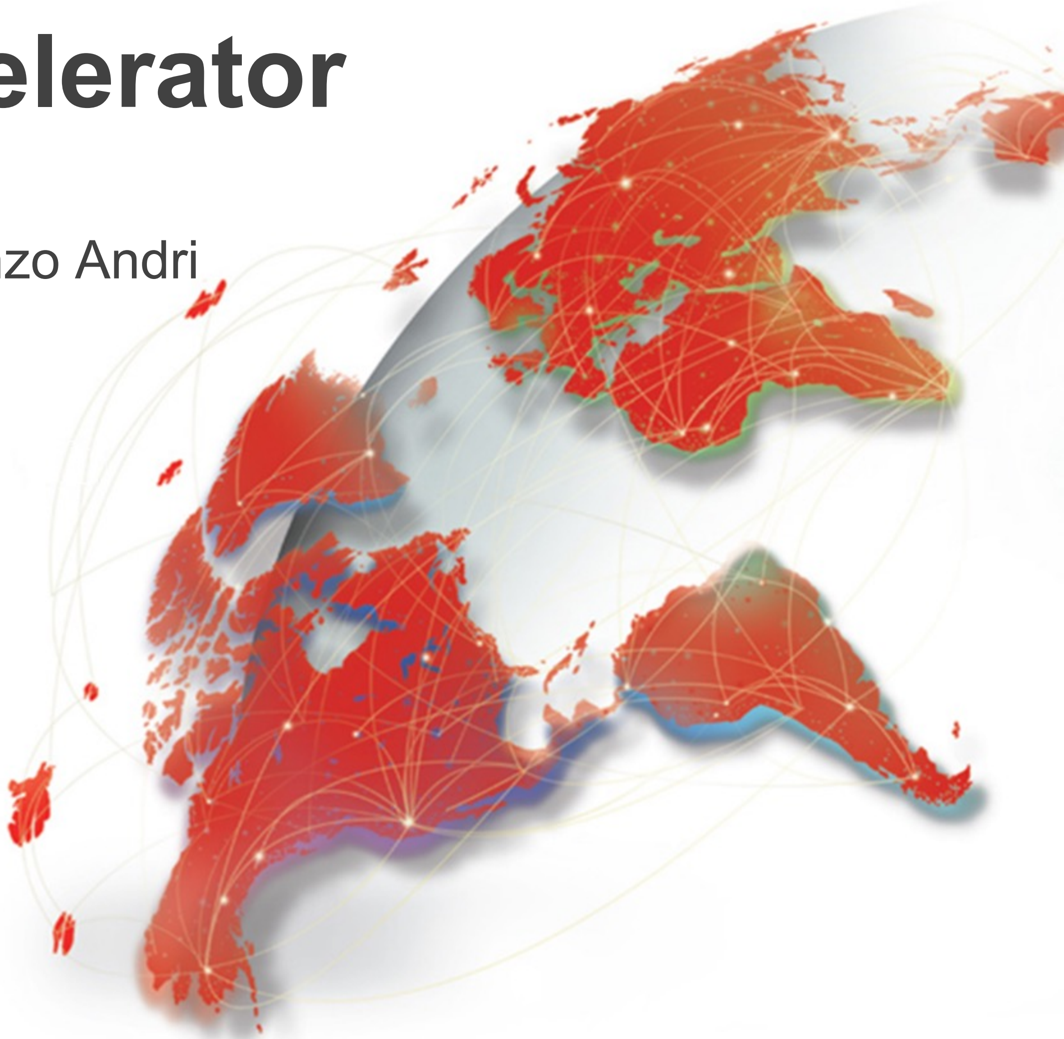
# MA Project: MADDNESS Accelerator

Jannis Schönleber, Matteo Perotti, Lukas Cavigelli, Renzo Andri



# Motivation

- **Matrix multiplications are key:**
  - Machine learning
  - Numerical solvers
- **New method for approximate matrix multiplications ("MADDness"):** ICML'21, arXiv

**Multiplying Matrices Without Multiplying**

**Davis Blalock** [1 2] **John Guttag** [2]

[1]MosaicML, San Francisco, CA, USA [2]MIT CSAIL

  - code available: python (here, here) and C (here, with AVX2)
- **Convenient for many applications:**
  - not just strong quantization (fewer issues): continuous, more stable/less noisy
  - applicable to anything approximate or with iterative refinement
  - more fine-grained & flexible than quantization

# MADDness: Method

- **What does MADDness solve?**
  - $A \in \mathbb{R}^{N\times D}, B \in \mathbb{R}^{D\times M}, \;\; N \gg D \geq M$
  - Given compute time budget $\tau$,

    find g(.), h(.), f(.), $\alpha, \beta$, such that

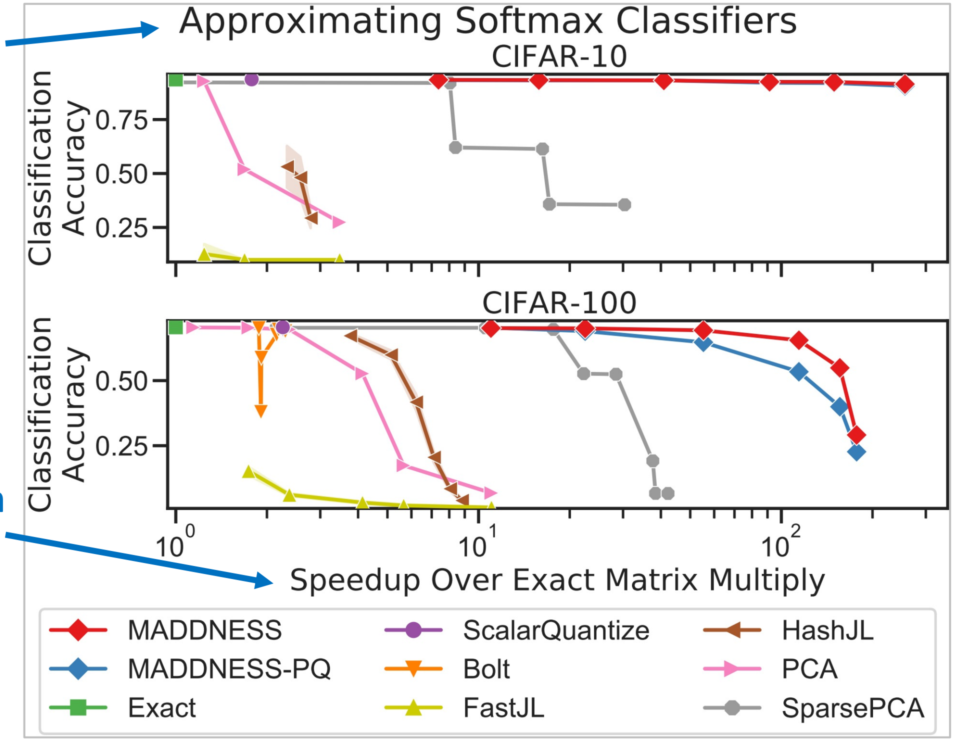
    $\left\| \alpha f\big(g(A), h(B)\big) + \beta - AB \right\|_F < \varepsilon(\tau) \|AB\|_F$
- **MADDNESS:**
  - makes use of $B$ matrix being fixed ("weights")
  - input for inference:
    - (1) matrix A,
    - (2) tensor of look-up tables $T$ computed from $B$
  - $(AB)_{n,m} \approx \alpha f\big(g(A), h(B)\big)_{n,m} + \beta = \alpha \sum_{c=1}^{C} T_{m,c,k} + \beta$

    with $k = g^{(c)}(a_n)$

not a full DNN, just last layer(s)!

use VGG-like DNN for FE, then learn 512→1000 linear classifier, then approximate this

based on 1-thread CPU implementation




**Algorithm 1** MADDNESSHASH

```
Input: vector x, split indices j^1, ..., j^4, split thresholds v^1, ..., v^4
i ← 1                                   // node index within level of tree
for t ← 1 to 4 do
    v ← v_i^t   // lookup split threshold for node i at level t
    b ← x_{j^t} ≥ v ? 1 : 0               // above split threshold?
    i ← 2i − 1 + b                        // assign to left or right child
end for
return i
```



ETH zürich

HUAWEI

# Compute Workload

- **Formula:**

$$(AB)_{n,m} \approx \alpha f\big(g(A), h(B)\big)_{n,m} + \beta = \alpha \sum_{c=1}^{C} T_{m,c,k} + \beta \quad \text{with } k = g^{(c)}(a_n)$$

with $A \in \mathbb{R}^{N\times D}$, $B \in \mathbb{R}^{D\times M}$, $C$ the #codebooks, $K$ the #learned prototypes

- **Computation & Lookups**
  - $g(A)$: $\Theta(NC)$
  - $h(B)$: $\Theta(MKCD)$, typically offline
  - $f(\cdot,\cdot)$: $\Theta(NCM)$, with $C$ table lookups for each output value ($M$ cols, $N$ rows)
  - overall: $\Theta\big(MC(KD+N)\big) \xrightarrow{K=16,\, N\gg D} \Theta(NCM)$
- **Storage** (based on the example in the paper)
  - VGG-style, CIFAR-100, classifier: D=512, K=16, M=100, $C \in \{16, 32, 64\}$
  - $T \in \mathbb{R}^{M\times C\times K}$, thus in the order of 100kB (C=32, float16)
  - reference: the “normal” linear layer would also use $D \cdot M$ weights, i.e. 100kB

# Project Plan (MA: 26 weeks)

- **Apply Maddness to 1x1 conv layers in ResNet-50 [3-4 weeks]**
  - get familiar with method
  - define/specify baseline quantized NN
  - replace some conv1x1 layers and find good parameters
  - collect statistics on access patterns, parameters (tables sizes, hash params, locality)
- **System & device-level architecture exploration [4-6 weeks]**
  - analyze perf/power/area trade-offs and flexibility for fundamentally different architecture, tune the memory hierarchy & interconnect, and *consider load balancing* (possibly with software interaction):
    - fully-specialized systolic arrays (also chiplets, multi-chip scale-out)
    - a PULP-style (MemPool-style?) architecture with ISA extension and/or per-cluster accelerator(s)
  - answer fundamental implementation questions: Flexible or fixed size matmul? Store values or hashes for FMs?
  - identify key building blocks & refine their PPA estimates
- **Implement (RTL & backend), integrate, verify, evaluate the most promising architecture [6-7 weeks]**
- **Develop basic software to feed & control the circuit [4-5 weeks]**
- **Map a full ResNet-50 or a recent BERT or MobileNetV3 model, think about other applications [4-5 weeks]**
- **Report, presentation, final experiments, code clean-up [3 weeks]**

ZURICH **RESEARCH** CENTER

# Weekly Update 0

- **What happened**
  - Beat COVID ☺ started working on Sunday
  - Git setup + remote setup (e.g. Badile, Picos, condor…)
  - Python code cleanup 17k -> 1.3k (right now) -> 800 LOC possible

# Weekly Update 0

- **Python Cleanup**
  - Merged all madness code
  - Pylint + Pytest + Github CI integration for easy and fast refactoring

- **GOAL:**
  - Close to our + paper mathematical formulation (bolt code far away)
  - Have a good structure for exploration + collect statistics on access patterns, parameters (tables sizes, hash params, locality)

- **Mathematical formulation:**

$(AB)_{n,m} \approx \alpha f\big(g(A), h(B)\big)_{n,m} + \beta = \alpha \sum_{c=1}^{C} T_{m,c,k} + \beta$ with $k = g^{(c)}(a_n)$

with $A \in \mathbb{R}^{N \times D}$, $B \in \mathbb{R}^{D \times M}$, $C$ the #codebooks, $K$ the #learned prototypes

# Weekly Update 0

```python
XtX = XtX.astype(np.float64)
XtY = XtY.astype(np.float64)

# preconditioning to avoid numerical issues (seemingly unnecessary, but
# might as well do it)
# scale = 1. / np.std(XtX)
if precondition:

    # # pretend cols of X were scaled differently
    # xscales = np.linalg.norm(XtX, axis=0) + 1e-20
    # mulby = (1. / xscales)
    # XtX *= mulby * mulby
    # XtY *= mulby.reshape(-1, 1)

    # yscales = np.linalg.norm(XtY, axis=1) + 1e-20
    # yscales = np.linalg.norm(XtY, axis=0) + 1e-20
    # yscales = yscales.reshape(-1, 1)

    # xscales = np.mean(np.linalg.norm(XtX, axis=0))
    # xscales = 7
    # xscales = 1

    # XtY *= (1. / yscales)
    # XtY *= (1. / yscales.reshape(-1, 1))

    # scale = 1. / len(X_enc)
    scale = 1.0 / np.linalg.norm(XtX, axis=0).max()
    XtX = XtX * scale
    XtY = XtY * scale

# W = np.linalg.solve(XtX, XtY)
W, _, _, _ = np.linalg.lstsq(XtX, XtY, rcond=None)  # doesn't fix it

# W, _, _, _ = np.linalg.lstsq(X_bin, Y, rcond=None)

# import torch
# import torch.nn.functional as F
# import torch.optim as optim

# def _to_np(A):
#     return A.cpu().detach().numpy()
```

```python
def optimal_split_val(
    self, X, dim, possible_vals=None, X_orig=None, return_possible_vals_losses=False
):
    if self.N < 2 or self.point_ids is None:
        if return_possible_vals_losses:
            return 0, 0, np.zeros(len(possible_vals), dtype=X.dtype)
        return 0, 0
    my_idxs = np.asarray(self.point_ids)
    if X_orig is not None:
        X_orig = X_orig[my_idxs]
    return optimal_split_val(
        X[my_idxs],
        dim,
        possible_vals=possible_vals,
        X_orig=X_orig,
        return_possible_vals_losses=return_possible_vals_losses,
    )
```

```python
# I honestly don't know why this is the formula, but wow
# does it work well
bias = self.number_of_codebooks / 4 * np.log2(self.upcast_every)
dists -= int(bias)
```

# Weekly Update 0

- **Next steps**
  - Finish python refactoring
  - C++ code refactoring
  - Build ResNet-50 Model with Tensorflow or PyTorch and integrate Maddness

- **Ideas & Discussion**
  - Current project/git repo name (uninspired) is halutmatmul
  - Is it possible to find a general hash/encoding (splits, thresholds) function with C=.. K=.. for AI application with batch normalized or normalized data? We would not need to learn those.

# Research Questions

1. Can this be used for DNN inference (ResNet, BERT [more sensitive, maybe focus on matmuls with Q, K, V])? What are typical parameters? What are typical access patterns?
2. What is the most efficient (energy, throughput/area) hardware architecture we can think of for this? What is a suitable memory hierarchy, memory types?
3. What is the more general design/trade-off space? Are there hard cliffs? Can we make it flexible/scalable/more efficient/… as a system?
4. What is a minimal change that gets most of the benefits (e.g., ISA extension)?
5. How efficient is it with an actual implementation?
6. Are there more good applications/use-cases? Also training? Higher accuracy with quantization-aware training?