FPGA-accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off

Kaan Kara, Dan Alistarh, Gustavo Alonso, Onur Mutlu, Ce Zhang
What is the most efficient way of training dense linear models on an FPGA?

• FPGAs can handle floating-point, but they are certainly not the best choice. Yet!

→ We have tried: On par with a 10-core Xeon, not a clear win.

• How about some recent developments in the machine research?

  In theory

  Low-precision data leads to EXACTLY the same end-result.
Stochastic Gradient Descent (SGD)

\[
\text{while not converged do}
\quad \text{for } i \text{ from 1 to } M \text{ do}
\quad \quad g_i = (a_i \cdot x - b_i) a_i
\quad \quad x \leftarrow x - \gamma g_i
\quad \text{end}
\quad \text{end}
\]

\[
A_{M \times N} \quad x_{N \times 1} = b_{M \times 1}
\]

Data Source
- DRAM, SSD, Sensor

Data
- \( A_r \)

Computation
- CPU, GPU, FPGA

Gradient:
- \( \text{dot}(A_r, x)A_r \)

Memory
- CPU Cache, DRAM, FPGA BRAM

Data Set, Model, Labels
Advantage of Low-Precision Data + FPGA

Compress!

\[
\text{while not converged do}
\begin{align*}
\text{for } i \text{ from 1 to } M & \text{ do} \\
& g_i = (a_i \cdot x - b_i)a_i \\
& x \leftarrow x - \gamma g_i \\
\text{end}
\end{align*}
\]

Data Set $A_{N \times M}$
Model $x_{N \times 1}$
Labels $b_{M \times 1}$

Data Source
DRAM, SSD, Sensor

Computation
CPU, GPU, FPGA

Memory
CPU Cache, DRAM, FPGA BRAM
Key Takeaway

- Using an FPGA, we can increase the **hardware efficiency** while maintaining the **statistical efficiency** of SGD for dense linear models.

In practice, the system opens up a multivariate trade-off space: Data properties, FPGA implementation, SGD parameters...
Outline for the Rest…

1. Stochastic quantization. Data layout
2. Target platform: Intel Xeon+FPGA
3. Implementation: From 32-bit to 1-bit SGD on FPGA.
4. Evaluation: Main results. Side effects.
5. Conclusion
Stochastic Quantization [1]

Naive solution: Nearest rounding (=1)
=> Converge to a different solution.

Stochastic rounding: 0 with probability 0.3, 1 with probability 0.7
=> Expectation remains the same. Converge to the same solution.

\[
\min_x \frac{1}{2} \sum_r (A_r \cdot x - b_r)^2
\]

Gradient

\[
g_i = (a_i \cdot x - b_i) a_i
\]

\[
g_i = (Q_1(a_i) \cdot x - b_i) Q_2(a_i)
\]

We need 2 independent samples

...and each iteration of the algorithm needs fresh samples.

Before the FPGA: The Data Layout

Currently slow!

Quantize to 4-bit. We need 2 independent quantizations!

4x compression!

1 quantization index

For each iteration
A new index!
Target Platform

96GB Main Memory

~30 GB/s

Mem. Controller

Caches

Intel Xeon CPU

6.5 GB/s QPI

QPI Endpoint

Accelerator

Altera Stratix V

Ivy Bridge, Xeon E5-2680 v2
10 cores, 25 MB L3

Intel Xeon+FPGA
Full Precision SGD on FPGA

32-bit floating-point: 16 values
Processing rate: 12.8 GB/s
Scale out the design for low-precision data!

How can we increase the internal parallelism while maintaining the processing rate?
Challenge + Solution

8-bit: 32 values Q8
4-bit: 64 values Q4
2-bit: 128 values Q2
1-bit: 256 values Q1

=> Scaling out is not trivial!

1) We can get rid of floating-point arithmetic.
2) We can further simplify integer arithmetic for lowest precision data.
Trick 1: Selection of quantization levels

1. Select the lower and upper bound \([L, U]\)
2. Select the size of the interval \(\Delta\)

\[\blackrightarrow\text{All quantized values are integers!}\]
This is enough for Q4 and Q8

8-bit: 32 values    Q8
4-bit: 64 values    Q4
2-bit: 128 values   Q2
1-bit: 256 values   Q1
Trick 2: Implement Multiplication using Multiplexer

Diagram:

- **In[31:0]**
- **Q2value[1:0]**
- **Out[31:0]**

- **Q2value normalized to [0,2] or [-1,1]**

- **Q2 multiplier**
  - **00**
  - **01**
  - **10**
  - **<<1**

- **Q1 multiplier**
  - **0**
  - **1**

- **Out[31:0]**
- **Result[31:0]**
Support for all Qx

Circuit for Q8, Q4 and Q2 SGD
Processing rate: 12.8 GB/s

Circuit for Q1 SGD
Processing rate: 6.4 GB/s
Support for all Qx

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Logic</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>38%</td>
<td>12%</td>
<td>7%</td>
</tr>
<tr>
<td>Q8</td>
<td>35%</td>
<td>25%</td>
<td>6%</td>
</tr>
<tr>
<td>Q4</td>
<td>36%</td>
<td>50%</td>
<td>6%</td>
</tr>
<tr>
<td>Q2, Q1</td>
<td>43%</td>
<td>1%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Circuit for Q8, Q4 and Q2 SGD
Processing rate: 12.8 GB/s

Circuit for Q1 SGD
Processing rate: 6.4 GB/s
## Data sets for evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th># Features</th>
<th># Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>60,000</td>
<td>780</td>
<td>10</td>
</tr>
<tr>
<td>GISETTE</td>
<td>6,000</td>
<td>5,000</td>
<td>2</td>
</tr>
<tr>
<td>EPSILON</td>
<td>10,000</td>
<td>2,000</td>
<td>2</td>
</tr>
<tr>
<td>SYN100</td>
<td>10,000</td>
<td>100</td>
<td>Regression</td>
</tr>
<tr>
<td>SYN1000</td>
<td>10,000</td>
<td>1,000</td>
<td>Regression</td>
</tr>
</tbody>
</table>

Following the Intel legal guidelines on publishing performance numbers, we would like to make the reader aware that results in this publication were generated using preproduction hardware and software, and may not reflect the performance of production or future systems.
SGD on Synthetic100, 4-bit works

SGD on Synthetic1000, 8-bit works
1-bit also works on some data sets

SGD on MNIST, Digit 7
1-bit works

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Accuracy</th>
<th>Training Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>float CPU-SGD</td>
<td>85.82%</td>
<td>19.04 s</td>
<td></td>
</tr>
<tr>
<td>1-bit FPGA-SGD</td>
<td>85.87%</td>
<td>2.41 s</td>
<td></td>
</tr>
</tbody>
</table>

MNIST multi-classification accuracy

Time (s) vs. Training Loss

- float CPU 1-thread
- float CPU 10-threads
- Q1 FPGA
- float FPGA

10.6x
Naïve Rounding vs. Stochastic Rounding

Stochastic quantization results in unbiased convergence.
Effect of Step Size

Large step size + Full precision vs. Small step size + Low precision
Conclusion

• Highly scalable, parametrizable FPGA-based stochastic gradient descent implementations for doing linear model training.
• Open source: www.systems.ethz.ch/fpga/ZipML_SGD

Key Takeaways:

1. **The way to train linear models on FPGA should be through the usage of stochastically rounded, low-precision data.**
2. **Multivariate trade-off space: Precision vs. end-to-end runtime, convergence quality, design complexity, data and system properties.**
Backup

Why do dense linear models matter?

- Workhorse algorithm for regression and classification.
- Sparse, high dimensional data sets can be converted into dense ones.

Why is training speed important?

- Often a human-in-the-loop process.
- Configuring parameters, selecting features, data dependency etc.
Effect of Reusing Indexes

In practice 8 indexes are enough!