A Fully Pipelined Kernel Normalised Least Mean Squares Processor For Accelerated Parameter Optimisation

Nicholas J. Fraser, Duncan J.M. Moss, JunKyu Lee, Stephen Tridgell, Craig T. Jin and Philip H.W. Leong
September 3, 2015

CELab, University of Sydney, Australia
Contributions:

• A fully pipelined core implementing KNLMS.
• A complete PCIe system.
• Large speedups over previous designs.
• Floating point and *fused* arithmetic designs - details in the paper!
Introduction: Motivation & Aims

Machine learning - creating models to fit data.

Two common problems arise:

• How do we decide which algorithm to use?
• Given an algorithm, how do we configure it?
Parameter selection can be the difference between a good and bad models.

Figure 1: The accuracy of KNLMS while varying a single parameter.
**Background: Kernel Methods**

KNLMS model:
- A dictionary, $\mathcal{D}$.
- A vector of weights, $\alpha$.
- A kernel function, $\kappa(x_i, x_j)$.

Prediction calculation (given $x_t$):

$$ \tilde{y}_t = \sum_{i=0}^{N} \alpha_i \kappa(x_t, \tilde{x}_i) = k^T \alpha. $$

The expressions are recursive - this creates a dependency problem!
Avoid the dependency problem by reordering the loop.

```c
for (parameters) {
    for (examples) {
        learn_model();
    }
}
```

```c
for (examples) {
    for (parameters) {
        learn_model();
    }
}
```
The core calculates a non-recursive portion of KNLMS.
\[ \kappa(x_t, \tilde{x}_i) = e^{-\gamma \|x_t - \tilde{x}_i\|_2^2}, \quad \forall i \in \{1, \cdots, N\} \]

\[ D_t = \begin{cases} [D_{t-1}; x_t] & \text{if } \mu < \mu_0 \\ D_{t-1} & \text{otherwise} \end{cases} \]

\[ \alpha_t = \alpha_{t-1} + \frac{\eta}{\epsilon + \|k\|_2^2} (y_t - \tilde{y}_t)k \]

\( \sim 210 \) pipeline stages were needed to achieve \( \sim 300\text{MHz} \).
On each clock cycle, a different set of parameters is passed in.
The core achieves speedups of $300 \times / 2,800 \times$ over a CPU (C) and a vector processor respectively.

The core is capable of learning $\sim 210$ models at $\sim 160$GFLOPS.
**RESULTS: SCALABILITY**

- VC707 (Virtex 7): \( N=16, M=8 \). \( N \) = dictionary entries, \( M \) = feature length.
- Virtex Ultrascale+: \( N=64, M=8 \) (estimated).

![Graphs showing LUT Usage, DSP Usage, and Latency with varying \( N \) or \( M \).](image)
• Demonstration of a fully pipelined machine learning core:
  • The core achieves $\sim 300\times /\sim 2,800\times$ speedups over a CPU and a previous design.
  • A 210 stage pipeline core achieves 160GFLOPS.
  • PCI system achieves $\sim 70\times /\sim 660\times$ speedups over a CPU and a previous design.

• Floating point and fused arithmetic investigated - details in the paper!

• We hope this design enables embedded and real-time applications for machine learning.
Future work includes:

- Further investigation of precision tradeoffs.
- Reducing the design latency.
- Implementing other machine learning algorithms.
- Using other platforms, such as GPUs and heterogeneous architectures.
Thank you. Any questions?


Appendices
Figure 2: Comparison of the ability of LMS and KNLMS to learn $f(x) = \text{sinc}(x)$.
**Table 1:** Computational complexity of some different machine learning methods when used in an online setting. Note that usually $m \leq n \ll N$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Computational Cost</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS [Widrow and Hoff, 1960]</td>
<td>$\mathcal{O}(m)$</td>
<td>(Simple)</td>
</tr>
<tr>
<td><strong>KNLMS [Richard et al., 2009]</strong></td>
<td>$\mathcal{O}(nm)$</td>
<td>(Modest)</td>
</tr>
<tr>
<td>KRLS [Engel et al., 2004]</td>
<td>$\mathcal{O}(nm + n^2)$</td>
<td>(Moderate)</td>
</tr>
<tr>
<td>SVM [Platt et al., 1998]</td>
<td>$\sim (N) \rightarrow (N^{2.2})$</td>
<td>(High)</td>
</tr>
</tbody>
</table>
Some kernel functions include:

- The polynomial kernel: \( \kappa(x_i, x_j) = (x_i^T x_j + c)^d \).
- The Gaussian kernel: \( \kappa(x_i, x_j) = e^{-\gamma \|x_i - x_j\|_2^2} \).
- For the Gaussian kernel: a measure of “similarity” between input vectors. The Gaussian kernel was used in this work.
Given a new input/output pair, \(\{x_t, y_t\}\), a model update is calculated as follows:

1. Evaluate \(\kappa\) between \(x_t\) and each entry of \(D_{t-1}\), creating a *kernel vector*, \(k\).
2. If \(\max(|k|) < \mu_0\), add \(x_t\) to the dictionary, producing \(D_t\).
3. Update the weights using:

\[
\alpha_t = \alpha_{t-1} + \frac{\eta}{\epsilon + k^T k} (y_t - k^T \alpha_{t-1}) k.
\]

How can we chose \(\kappa\), \(\mu_0\), \(\eta\) and \(\epsilon\)? We must do a *parameter search*. 
1. Create a dataflow graph (with no loops!) from a non-recursive section of the KNLMS algorithm, i.e. the update step. We call this module the \textit{forward path}.

2. Map the dataflow graph to hardware.

3. Pipeline the hardware to achieve a desired throughput.

4. Connect an external scheduler to process parallel jobs.

A benefit of designing the hardware this way, is that the pipelining can be achieved using high level tools, such as Vivado HLS or Altera DSP Builder.
The core calculates a non-recursive portion of the KNLMS algorithm.
The core can be pipelined arbitrarily to achieve virtually any desired clock speed.
The complete PCI-System utilises a FSM to iterate through parameters in order to fill the KNLMS core pipeline.
**Figure 3:** Dataflow diagram of a kernel module.

**Figure 4:** Dataflow diagram of the α update module.
• The design was made using Vivado HLS.
• Two variants of the core were made, one which used single precision floating point arithmetic (float), the other used a combination of fixed and single precision floating point (fused).
• A PCI-based system implementation was created using RIFFA 2.0 [Jacobsen et al., 2012]. The system implementation includes optimizations specific to a parameter search problem.
• The design was compared to: KAFBOX [Van Vaerenbergh, 2012] (MATLAB), an optimised C implementation, a naïve Vivado HLS and a previous microcoded vector processor implementation [Pang et al., 2013].
### Table 2: Comparison of online kernel method implementations. Note that $N = 16$ and $M = 8$.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Algorithm</th>
<th>DSPs</th>
<th>Freq (MHz)</th>
<th>Time (ns)</th>
<th>Slowdown rel. to Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve KNLMS Float</td>
<td>KNLMS</td>
<td>12</td>
<td>96.7</td>
<td>7,829</td>
<td>2,462</td>
</tr>
<tr>
<td>CPU (C) KNLMS</td>
<td>KNLMS</td>
<td>-</td>
<td>3,600</td>
<td>940</td>
<td>296</td>
</tr>
<tr>
<td>CPU (KAFBOX) KNLMS</td>
<td>KNLMS</td>
<td>-</td>
<td>3,600</td>
<td>73,655</td>
<td>23,162</td>
</tr>
<tr>
<td>Pang et al. [2013]</td>
<td>SW-KRLS</td>
<td>30</td>
<td>237</td>
<td>9,000</td>
<td>2,830</td>
</tr>
</tbody>
</table>
### Table 3: Comparison of online kernel method implementations. Note that $N = 16$ and $M = 8.$

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Algorithm</th>
<th>DSPs</th>
<th>Freq (MHz)</th>
<th>Time (ns)</th>
<th>Slowdown rel. to System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>KNLMS</td>
<td>12</td>
<td>96.7</td>
<td>7,829</td>
<td>576</td>
</tr>
<tr>
<td>Float</td>
<td>KNLMS</td>
<td>1267</td>
<td>314</td>
<td>3.18</td>
<td>0.23</td>
</tr>
<tr>
<td>System</td>
<td>KNLMS</td>
<td>691</td>
<td>250</td>
<td>13.6</td>
<td>1</td>
</tr>
<tr>
<td>CPU (C)</td>
<td>KNLMS</td>
<td>-</td>
<td>3,600</td>
<td>940</td>
<td>69</td>
</tr>
<tr>
<td>CPU (KAFBOX)</td>
<td>KNLMS</td>
<td>-</td>
<td>3,600</td>
<td>73,655</td>
<td>1703</td>
</tr>
<tr>
<td>Pang et al. [2013]</td>
<td>SW-KRLS</td>
<td>30</td>
<td>237</td>
<td>9,000</td>
<td>661</td>
</tr>
</tbody>
</table>
Table 4: Area utilisation of different designs obtained from synthesis. Note the computational complexity of KNLMS is $O(mn)$.

<table>
<thead>
<tr>
<th>Type</th>
<th>$M$</th>
<th>$N$</th>
<th>LUTs</th>
<th>DSPs</th>
<th>$L$</th>
<th>$F_{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>2</td>
<td>16</td>
<td>77K</td>
<td>595</td>
<td>185</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>109K</td>
<td>819</td>
<td>196</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>307K</td>
<td>2163</td>
<td>218</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>23K</td>
<td>161</td>
<td>162</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>46K</td>
<td>319</td>
<td>177</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>95K</td>
<td>635</td>
<td>192</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>173K</td>
<td>1267</td>
<td>207</td>
<td>385</td>
</tr>
<tr>
<td>Fused</td>
<td>2</td>
<td>16</td>
<td>102K</td>
<td>494</td>
<td>161</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>119K</td>
<td>595</td>
<td>163</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>440K</td>
<td>1171</td>
<td>175</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>33K</td>
<td>101</td>
<td>131</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>64K</td>
<td>199</td>
<td>143</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>130K</td>
<td>395</td>
<td>155</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>247K</td>
<td>787</td>
<td>167</td>
<td>303</td>
</tr>
</tbody>
</table>