Deep Learning on FPGAs for Trigger and Data Acquisition

ACAT, 11-15 March 2019, Saas Fee, Switzerland

Javier Duarte, Sergo Jindariani, Ben Kreis, Ryan Rivera, Nhan Tran (Fermilab)
Jennifer Ngadiuba, Maurizio Pierini, Vladimir Loncar (CERN)
Edward Kreinar (Hawkeye 360)
Phil Harris, Song Han, Dyan Rankin (MIT)
Zhenbin Wu (University of Illinois at Chicago)
Sioni Summers (Imperial College London)
Giuseppe Di Guglielmo (Columbia University)
Motivation
The challenge: triggering at (HL-)LHC
The challenge: triggering at (HL-)LHC

Extreme bunch crossing frequency of 40 MHz → extreme data rates O(100 TB/s)

“Triggering” = filter events to reduce data rates to manageable levels
The challenge: triggering at (HL-)LHC

Extreme bunch crossing frequency of 40 MHz → extreme data rates $O(100 \text{ TB/s})$

“Triggering” = filter events to reduce data rates to manageable levels

Squeeze the beams to increase data rates
→ multiple pp collisions per bunch crossing (pileup)

2016: $<\text{PU}> \sim 20-50$
2017 + Run 3: $<\text{PU}> \sim 50-80$
HL-LHC: 140-200

CHALLENGE: maintain physics in increasingly complex collision environment
→ untriggered events lost forever!

Sophisticated techniques needed to preserve the physics!
A typical trigger system

Triggering typically performed in multiple stages @ ATLAS and CMS

L1 Trigger

40 MHz

100 kHz

High-Level Trigger

1 kHz
1 MB/evt

Offline

Absorbs 100s Tb/s

Trigger decision to be made in $O(\mu s)$

Latencies require all-FPGA design

Computing farm for detailed analysis of the full event

Latency $O(100 \text{ ms})$

For HL-LHC upgrade: latency and output rates will increase by $\sim 3$ (ex: for CMS 3.8 $\rightarrow$ 12.5 $\mu$s @ L1)
The latency landscape @ LHC

100 ms

1 s

ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ….

ML algorithms used offline for

* improving Higgs mass resolution with particle energy regression
* enhancing signal/background discrimination

ex, Higgs discovery

CMS

m_{\gamma\gamma} (GeV)

S/(S+B) weighted events / GeV

19.7 fb^{-1} (8 TeV) + 5.1 fb^{-1} (7 TeV)

ML algorithms used offline for

* improving Higgs mass resolution with particle energy regression
* enhancing signal/background discrimination

Ex, Higgs discovery
The latency landscape @ LHC

Exploration of ML algorithms in low-latency, real-time processing has just begun!

What can we do in ~1 us on one FPGA?

ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ....
hls4ml

Machine Learning & FPGAs
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain array of logic cells embedded with DSPs, BRAMs, etc.

High speed input/output to handle the large bandwith

Support highly parallel algorithm implementations

Low power (relative to CPU/GPU)

Digital Signal Processors (DSPs): logic units used for multiplications

Random-access memories (RAMs): embedded memory elements

Flip-flops (FF) and look up tables (LUTs) for low level operations
How are FPGAs programmed?

Hardware Description Languages

HDLs are programming languages which describe electronic circuits.

High Level Synthesis

generate HDL from more common C/C++ code
pre-processor directives and constraints used to optimize the timing

*drastic decrease in firmware development time!*

We use here Xilinx Vivado HLS [*], but plan also Intel/Altera Quartus HLS

Neural network inference

\[ x_n = g_n(W_{n,n-1}x_{n-1} + b_n) \]

- **Activation function**: ReLU
- **Multiplication** precomputed and stored in BRAMs
- **Addition** in DSPs and logic cells

Input layer: 16 inputs
- Layer \( L_1 \): 64 nodes, activation: ReLU
- Layer \( L_2 \): 32 nodes, activation: ReLU
- Layer \( L_3 \): 32 nodes, activation: ReLU
- Output layer: 5 outputs, activation: SoftMax

Number of multiplications:

\[ N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n \]
Neural network inference

\[
x_n = g_n(W_{n,n-1}x_{n-1} + b_n)
\]

- \(x_n\): input layer
- \(g_n\): activation function
- \(W_{n,n-1}\): multiplication
- \(b_n\): addition
- \(x_{n-1}\): precomputed and stored in BRAMs
- \(M\): hidden layers
- \(N\): output layer

\[
N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n
\]

- 16 inputs
- 64 nodes
- 5 outputs
- activation: SoftMax

How many resources? DSPs, LUTs, FFs?
Does the model fit in the latency requirements?
Case study: jet tagging

Study a multi-classification task: discrimination between highly energetic (boosted) $q$, $g$, $W$, $Z$, $t$ initiated jets

Just an illustrative example, lessons are generic! Might not be the best application, but a familiar one. ML in substructure is well-studied.

$t \rightarrow bW \rightarrow bqq$  
3-prong jet

$Z \rightarrow qq$  
2-prong jet

$W \rightarrow qq$  
2-prong jet

$q/g$ background  
no substructure and/or mass $\sim 0$

Signal: reconstructed as one massive jet with substructure

Jet substructure observables used to distinguish signal vs background [*]

Case study: jet tagging

- Study a 5-output multi-classification task: discrimination $q, g, W, Z, t$ initiated jets

- Fully connected neural network with **16 inputs**:
  
  - **mass** (Dasgupta et al., arXiv:1307.0007), **multiplicity, energy correlation functions** (Larkoski et al., arXiv:1305.0007)
  
  - expert-level features not necessarily realistic for L1 trigger, but the lessons here are generic

AUC = area under ROC curve
(100% is perfect, 20% is random)
high level synthesis for machine learning

Implemented an user-friendly and automatic tool to develop and optimize FPGA firmware design for DL inference:

- reads as input models trained with standard DL libraries
- uses Xilinx HLS software (accessible to non-expert, engineers resource not common in HEP)
- comes with implementation of common ingredients (layers, activation functions, binary NN ….)

[hls4ml](https://hls-fpga-machine-learning.github.io/hls4ml/)

Efficient NN design for FPGAs

FPGAs provide huge flexibility

*Performance depends on how well you take advantage of this*

With hls4ml package we have studied/optimized the FPGA design through:

- **compression**: reduce number of synapses or neurons
- **quantization**: reduces the precision of the calculations (inputs, weights, biases)
- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources

Constraints:

- Input bandwidth
- FPGA resources
- Latency

FPGAs provide huge flexibility. Performance depends on how well you take advantage of this.

FPGA project designing

NN training
Efficient NN design: compression

- Iterative approach:
  - train with \textbf{L1 regularization} (loss function augmented with penalty term):
    \[
    L_\lambda(\vec{w}) = L(\vec{w}) + \lambda||\vec{w}_1||
    \]
  - sort the weights based on the value relative to the max value of the weights in that layer

---

1st iteration

Train with L1
Efficient NN design: compression

- Iterative approach:
  - train with **L1 regularization** (loss function augmented with penalty term):
    \[ L_\lambda(\vec{w}) = L(\vec{w}) + \lambda ||\vec{w}_1|| \]
  - sort the weights based on the value relative to the max value of the weights in that layer
  - prune weights falling below a certain percentile and retrain
Efficient NN design: compression

Prune and repeat the training for 7 iterations

1st iteration

2nd iteration

7th iteration
Efficient NN design: compression

Prune and repeat the training for 7 iterations

→ 70% reduction of weights and multiplications w/o performance loss
Efficient NN design: quantization

- In FPGAs use **fixed point data types**
  → less resources and latency than 32-bit floating point
- Quantify the performance of the classifier with the AUC
- Expected AUC = AUC achieved by 32-bit floating point inference of the neural network

\[ \text{ap\_fixed<width,integer>} \]

0101.1011101010

- **integer**
- **fractional**
- **width**

**Scan integer bits**
Fractional bits fixed to 8

**Scan fractional bits**
Integer bits fixed to 6

**Full performance at 6 integer bits**

**Full performance at 8 fractional bits**

\[ \text{FPGA AUC / Expected AUC} \]
Efficient NN design: parallelization

- Trade-off between latency and FPGA resource usage determined by the parallelization of the calculations in each layer
- Configure the “reuse factor” = number of times a multiplier is used to do a computation

Reuse factor: how much to parallelize operations in a hidden layer
Resource usage: DSPs

- DSPs (used for multiplication) are often limiting resource
  - maximum use when fully parallelized
  - DSPs have a max size for input (e.g. 27x18 bits), so number of DSPs per multiplication changes with precision

70% compression ~ 70% fewer DSPs
Parallelization: Timing

![Graph showing latency and fixed-point precision for different reuse factors in an 8-layer pruned model on a Kintex Ultrascale FPGA.](image)

- Longer latency
- Each mult. used 6x
- Each mult. used 3x
- Fully parallel
- Each mult. used 1x
- More resources

---

04.06.2018

Jennifer Ngadiuba - hls4ml: deep neural networks in FPGAs
Synthesis vs implementation

- All previous results come from HLS synthesis estimates
- Final implementation gives actual resource usage and timing estimate
- For slightly smaller model:
  - FF/LUT - overestimated in most cases
  - DSPS - accurate below max width of multiplier input, overestimated above
hls4ml

Status and outlook
Large amount of ongoing development to expand tool capabilities

1. Adding/testing full support for **other NN architectures**
   - Conv1D/2D layers
   - LSTM/GRU
   - Graph Neural Network
   - Binary/Ternary dense networks
   - Pooling
   - Boosted Decision Trees

2. Adding/testing support for **large networks**
3. Extend hls4ml with **Altera support**
4. Tests feasibility of **CPU+FPGA co-processor** on Amazon cloud

**Stay tuned for updates!**

---

Use case of CNN @ LHC for jet image

JHEP 07 (2016) 069

Graph NN ex, tracking?

BDT
Summary

We introduced a new software/firmware package **hls4ml**

Automated translation of everyday machine learning inference into firmware in ~ minutes

Tunable configuration for optimization of your use case

This application is for single FPGA, ~1 us latency for L1 trigger or DAQ

Explore also applications for acceleration with CPU-FPGA co-processors for long latency trigger tasks

For more info


Backup
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1

Output: particle candidates

CMS-TDR-017, JINST 12 (2017) P10003
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1

Output: particle candidates

Machine learning

Triggers decision

CMS-TDR-017, JINST 12 (2017) P10003
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1

Output: particle candidates

Machine learning

THIS TALK!

Trigger decision

CMS-TDR-017, JINST 12 (2017) P10003
Muon reconstruction @ L1

First implementation of a ML algo for CMS L1 trigger on FPGAs [*]

A BDT is used to improve the momentum of muons in the forward region of the detector based on curvature angles in the magnetic fields $(\Delta \phi, \Delta \theta)$ and few other variables

Prediction of BDT for every possible input stored into pre-computed 1.2 GB Look-Up Table (LUT) on FPGA

Achieved reduction of background rates by factor 3 w/o efficiency losses

Usage of LUTs does not scale nicely with ML algo complexity → quickly use all resources

Can we improve this approach?

Jet substructure features

Jet substructure observables provide large discrimination power between these types of jets

mass, multiplicity, energy correlation functions, … (computed with FastJet [*])


These are expert-level features

Not necessarily realistic for L1 trigger
"Raw" particle candidates more suitable (to be studied next)
But lessons here are generic

One more case: H→bb discrimination vs W/Z→qq requires more “raw” inputs for b-tagging information
The **hls4ml** package

### Translation

```python
keras-to-hls.py -c keras-config.yml```

### Inputs

- **K** Keras

  - Keras HDF5

### Config

- **IOType**: parallelize or serialize
- **ReuseFactor**: how much to parallelize
- **DefaultPrecision**: inputs, weights, biases

---

Javier Duarte I

---

**Program Flow**

1. **python keras-to-hls.py -c keras-config.yml**

2. **KerasJson**: example-keras-model-files/KERAS_1layer.json
   ```
   KerasJson: example-keras-model-files/KERAS_1layer.json
   KerasH5: example-keras-model-files/KERAS_1layer_weights.h5
   ```
3. **OutputDir**: my-hls-test
   ```
   OutputDir: my-hls-test
   ```
4. **ProjectName**: myproject
   ```
   ProjectName: myproject
   ```
5. **XilinxPart**: xc7vx690tfsg1927-2
   ```
   XilinxPart: xc7vx690tfsg1927-2
   ```
6. **ClockPeriod**: 5
   ```
   ClockPeriod: 5
   ```
7. **IOType**: io_parallel
   ```
   IOType: io_parallel # options: io_serial/io_parallel
   ```
8. **ReuseFactor**: 1
   ```
   ReuseFactor: 1
   ```
9. **DefaultPrecision**: ap_fixed<18,8>
   ```
   DefaultPrecision: ap_fixed<18,8>
   ```

---

**Build Prj TCL**

- **fimware**
- **myproject_test.cpp**

---

Jennifer Ngadiuba - **hls4ml**: deep neural networks in FPGAs
The **hls4ml** package

### Build HLS project

```
# HLS4ML #
open_project -reset myproject_prj
set_top myproject
add_files firmware/myproject.cpp -cflags "-I[file normalize ../../nnet_utils]"
add_files -tb myproject_test.cpp -cflags "-I[file normalize ../../nnet_utils]"
add_files -tb firmware/weights
open_solution -reset "solution1"
set_part {xcku115-flvf1924-2-i}
csim_design
create_clock -period 5 -name default
csim_design
csynth_design
cosim_design -trace_level all
export_design -format ip_catalog
exit
```

Produce a firmware block in ~ minutes!
Parallelization: Timing

- **Initiation interval**: number of clocks before accepting new inference inputs
  - scales with reuse factor
  - for very low data precisions multiplications implemented through FFs and LUTs

- **Additional latency** introduced by reusing the multiplier

**Latency of layer m**

\[ L_m = L_{\text{mult}} + (R - 1) \times II_{\text{mult}} + L_{\text{activ}} \]
Other resources

Fairly linear increase with precision

Small percentage of total available

Spikes present at steep transitions in LUTs usage as artifacts of HLS synthesis

Not observed in implementation
Found also dependence on Vivado HLS version
Firmware implementation

- HLS synthesis estimate on resource usage are usually conservative
  - DSPs usage agree well below DSP precision transition (27 bit), deviations above due to further Vivado optimizations
  - FFs and LUTs overestimated by a factor 2-4
Firmware implementation

• Final implementation gives actual resource usage and timing estimate
  - how optimal is the HLS design?

• Power usage increases with precision, it goes down for less throughput (higher reuse factor)

• Implement a 1-layer NN, simply routing all firmware block’s inputs and outputs to FPGA available pins

• Timing target not always met (depends on the Vivado HLS version)

• HLS estimate on resource usage are conservative
  - DSPs usage agree well below DSP precision transition (27 bit), implementation does further optimization
  - FFs and LUTs overestimated by a factor 2-4
Extend `hls4ml` with Intel support

- **hls4ml** supports Xilinx FPGAs and software from beginning

- Currently working to extend the package to work with Intel/Altera Quartus HLS

  - work in progress: technical complications slowed us down (software licenses and installation, Quartus HLS version @ CERN, …)

- First results encouraging (based on emulation and to be confirmed with actual deployment on card)
goes to the cloud

- **Amazon Web Service** provides co-processor CPU+FPGA systems with Xilinx Virtex Ultrascale+ VU9P

- Xilinx SDAccel development environment allows the development/running of connected FPGA kernels and CPU processes
  - any FPGA application defined in HLS, OpenCL, or VHDL/Verilog can be accelerated

- **hls4ml** project only needs to be wrapped to provide specific I/O ports configuration for SDAccel to interface properly

- **Successfully accelerated 1D CNN example project on AWS F1**: 10 four-channel inputs, 3 convolutional layers, 2 dense layers, 5 outputs → *latency 116 ns*
DL acceleration on Intel FPGAs @ CERN

• **hls4ml** developed from start to target very low latency

• Latencies at HLT less strict allowing inference of much bigger networks → different firmware project design wrt L1 trigger application

• No need to reinvent the wheel…

• **Work in progress:** accelerate DL inference of predefined big networks through Intel softwares such as **OpenVino** to benchmark a specific physics case

  - perform studies on-site with Intel Arria 10 GX FPGA available at CERN
For long latency tasks
→ more time means
  More resource reuse (x1000)
  Bigger networks

ex: HLT or offline reconstruction @ LHC

Offload a CPU from the computational heavy parts to a FPGA “accelerator”

Increased computational speed of 10x-100x
Reduced system size of 10x
Reduced power consumption of 10x-100x

• **Amazon Web Service** provides cloud based system consisting of CPU/GPU/FPGAs

  - AWS F1 instances include up to 8 Xilinx Virtex Ultrascale+

• **Used hls4ml through SDAccel** to create a firmware implementation of 1D CNN

  - 5 layers with 10 four-channel inputs, latency of 116 ns
  - successfully run on an AWS F1 instance