The Compact Muon Solenoid Experiment
Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

14 November 2016 (v2, 15 November 2016)

Using MaxCompiler for High Level Synthesis of Trigger Algorithms

Sioni Paris Summers for the CMS Collaboration

Abstract

Firmware for FPGA trigger applications at the CMS experiment is conventionally written using hardware description languages such as Verilog and VHDL. MaxCompiler is an alternative, Java based, tool for developing FPGA applications which uses a higher level of abstraction from the hardware than a hardware description language. An implementation of the jet and energy sum algorithms for the CMS Level-1 calorimeter trigger has been written using MaxCompiler to benchmark against the VHDL implementation in terms of accuracy, latency, resource usage, and code size. A Kalman Filter track fitting algorithm has been developed using MaxCompiler for a proposed CMS Level-1 track trigger for the High-Luminosity LHC upgrade. The design achieves a low resource usage, and has a latency of 187.5 ns per iteration.

Presented at *TWEPP-16 Topical Workshop on Electronics for Particle Physics*

2 Using MaxCompiler for the High Level Synthesis of 3 Trigger Algorithms

4 **S. Summers,^a A. Rose,^a P. Sanders^b**

5 ^a*Imperial College London,*

6 *Blackett Laboratory, Imperial College London, Prince Consort Road, SW7 2AZ, UK*

7 ^b*Maxeler Technologies Ltd., 3-4 Albion Place, London, W6 0QT, UK*

8 *E-mail:* ss5310@imperial.ac.uk

9 ABSTRACT: Firmware for FPGA trigger applications at the CMS experiment is conventionally written
10 using hardware description languages such as Verilog and VHDL. MaxCompiler is an alternative,
11 Java based, tool for developing FPGA applications which uses a higher level of abstraction from
12 the hardware than a hardware description language. An implementation of the jet and energy
13 sum algorithms for the CMS Level-1 calorimeter trigger has been written using MaxCompiler to
14 benchmark against the VHDL implementation in terms of accuracy, latency, resource usage, and
15 code size. A Kalman Filter track fitting algorithm has been developed using MaxCompiler for a
16 proposed CMS Level-1 track trigger for the High-Luminosity LHC upgrade. The design achieves
17 a low resource usage, and has a latency of 187.5 ns per iteration.

¹Corresponding author.

1	Contents	
2	1 Introduction	1
3	1.1 MaxCompiler	1
4	2 CMS Level 1 Calorimeter Trigger	3
5	2.1 Jet and Energy Sum Algorithm Implementation	3
6	2.2 MaxJ Implementation	3
7	3 CMS Level 1 Track Trigger Demonstrator	4
8	3.1 Kalman Filter	5
9	3.1.1 MaxJ Implementation	6
10	3.1.2 Simulation	7
11	3.2 Control Flow	8
12	4 Summary	8

13 1 Introduction

14 High level synthesis (HLS) is an alternative way to develop applications for FPGAs, which are
15 typically programmed using hardware description languages (HDLs) such as VHDL and Verilog.
16 HLS tools enable development of applications from a procedural description, rather than from the
17 behavioural description written in HDL. The use of HLS may lead to shorter development times
18 and more maintainable code bases, given the different approach, and the abstraction of hardware
19 specifics from the application design. The requirements for any HLS tool in a High Energy Physics
20 (HEP) triggering application are: that the design must be testable and verifiable; that the utilization
21 of the FPGA resources should be efficient compared to a hand-written HDL equivalent; and that the
22 latency of the design meet the strict requirements for trigger systems. In this paper we explore the
23 use of MaxCompiler [1] in HEP trigger applications, using the Level-1 (L1) trigger of the Compact
24 Muon Solenoid (CMS) [2] experiment at the Large Hadron Collider (LHC).

25 MaxJ is the language used to describe algorithms for MaxCompiler. An extension of Java,
26 MaxJ allows the programmer to use a higher level of abstraction from the hardware than when using
27 HDLs. MaxCompiler compiles the code into output compatible with the FPGA vendor synthesis
28 tools. The synthesisable output takes the form of a combination of VHDL and vendor specific
29 components, such as for DSPs or RAMs.

30 1.1 MaxCompiler

31 A key feature of MaxCompiler is the scheduling of the design into a pipeline automatically. The
32 scheduler determines which operations must be in sequence with others, and which can be performed
33 in parallel. In the default mode each logic operation is followed by a register, forming the pipeline.

Figure 1. Dataflow graph for the simple design performing ' $z = x^2 + y$ ' with inputs x , y and output z . Operations are represented as nodes on the graph. The default latency for a DSP operation is 3 clock cycles - this is configurable - so the signal y is registered by the compiler for 3 clock cycles to maintain the synchronisation between x and y . The register is represented by the box labelled '3'.

1 When paths of different latency must meet, the shorter branch is pipelined so as to synchronise the
 2 two paths. A 'dataflow graph' of the design is produced, showing the scheduling of operations,
 3 and is used by the compiler for further optimisations of the design. Automatic pipelining simplifies
 4 design greatly, since the developer is freed from having to synchronise signals by hand. Meeting
 5 the requirement for low latency in a trigger algorithm may require the tuning of the number of
 6 registers in the datapath, which is supported by the language. This comes down to finding the
 7 balance between pipeline length and clock frequency.

8 Figure 1 shows an example dataflow graph for the simple design taking inputs x and y and
 9 outputting $z = x^2 + y$, produced using the following code:

```

10 DFEVar x = io.input("x", dfeInt(16)); // A 16 bit signed integer
11 DFEVar y = io.input("y", dfeInt(16));
12 DFEVar z = x * x + y;
13 io.output("z", z, z.getType());
  
```

14 In the source code the developer separates what part is intended for the FPGA and what runs on
 15 the CPU explicitly, using different objects for each. 'DFEVars' describe the objects on the FPGA,
 16 leaving the usual `int`, `float`, etc. types on the CPU for use at compile time only, for example to
 17 define constant values or create multiple instances of an operation. The following code inputs a
 18 variable `x`, and multiplies it with ten integer constants, in parallel, storing the results in the array `y`.
 19 The compiler will also map the operations to hardware efficiently, using shifts for multiplications
 20 by powers of 2, and simply registering `x` rather than multiplying by 1.

```

21 DFEVar x = io.input("x", dfeInt(16));
22 DFEVar[] y = new DFEVar[10];
23 for(int n = 0; n < 10; n++){
24     y[n] = x * n;
25 }
  
```

26 As a high level language, MaxJ provides some abstraction from the hardware implementation
 27 specifics of operations. Memories are defined more similarly to CPU applications: as a queue,
 28 FIFO, RAM, or ROM rather than by explicit instantiation of memory IP, or inferral by the synthesis
 29 tool from port definitions as with HDLs. Similarly, an operation such as $x \times y + z$ can be optimised to

1 use a DSP only by the compiler, with the source code simply stating ‘ $x * y + z$ ’. This abstraction
 2 also allows the developer some freedom in targeting applications to FPGAs from different vendors,
 3 since the compiler is tasked with mapping operations to hardware to a greater extent than in a
 4 conventional HDL application - notwithstanding different physical features in different FPGAs.

5 **2 CMS Level 1 Calorimeter Trigger**

6 The CMS Level-1 calorimeter trigger processes measurements from the muon detector and calorime-
 7 ters to trigger readout of the full detector. Within the calorimeter trigger the electromagnetic,
 8 hadronic, and hadronic forward calorimeters (ECAL, HCAL and HF respectively) provide trigger
 9 tower primitives with transverse energy and position at a reduced resolution.

10 A time multiplexing architecture is used, with the result that a single node processes all trigger
 11 towers in the event it receives. A first hardware layer sends the calorimeter data over optical fibre
 12 to a second layer over a period of several bunch crossings. All towers from one azimuthal ring at
 13 a particular pseudorapidity (η) slice are sent simultaneously, one from each side of the detector.
 14 There are 68 η slices, and 72 towers in a 2π ring in ϕ . Nodes in the second hardware layer find jets,
 15 e/γ and τ candidates, as well as the total and missing transverse energy in each event. The second
 16 hardware layer is comprised of MP7 boards, which feature a Xilinx Virtex-7 XC7VX690T FPGA
 17 for processing [4].

18 **2.1 Jet and Energy Sum Algorithm Implementation**

19 The two CMS L1 calorimeter trigger algorithms re-implemented using MaxCompiler are the jet
 20 and energy sum algorithms. The energy of all trigger towers is summed together, separately for
 21 electromagnetic and hadronic parts, to determine the total E_T in the event, and in parallel the vector
 22 sum is calculated to determine the missing energy. In the calorimeter trigger jets are defined as
 23 objects with an energy which is the sum of the energy in towers in a fixed 9×9 tower window
 24 around the centre, which is the highest energy tower in the window. Pileup is estimated locally
 25 from the energy in trigger towers neighbouring the window and subtracted from the jet. The twelve
 26 highest energy jets are passed to the output.

27 Due to the data ordering introduced by the time multiplexing, ϕ separation between towers
 28 corresponds to a separation across input links, while η separation corresponds to a separation in
 29 time. Implementing the jet algorithm therefore involves significant pipelining to make jets with
 30 energies spanning 9 towers in η . All the algorithms operate at a 240 MHz clock frequency.

31 **2.2 MaxJ Implementation**

32 Making efficient use of the resources of the chip in MaxJ required similar design patterns to the
 33 VHDL implementation. For example, neighbouring jet candidates have energy contributions from
 34 many of the same trigger towers. When computing the energy of many jet candidates in parallel,
 35 building the sum of tower energies in reusable arrangements makes for efficient use of the FPGA
 36 resources. In both languages this requires some particular constructions by the developer.

37 By passing data through each implementation we see that the two firmware outputs match well,
 38 as shown in figure 2. The MaxJ is seen to obtain results which are bit-identical to the VHDL. A small
 39 difference is present in the distribution of the azimuthal angles of jets. This is due to an internal

Figure 2. Output of the MaxJ and VHDL implementations of jet and energy sum algorithms on Monte Carlo data. A bit-exact matching is observed in the E_T of the jets and the transverse energy sums, and also in the additional output parameters of missing transverse energy, and jet η .

1 difference in the sorting, which in the MaxJ can lead to jets of equal energy and pseudo-rapidity
 2 emerging in the opposite order to the VHDL implementation. This difference would not affect the
 3 physics performance of the trigger.

4 Targeting the MP7 board, the MaxJ implementation uses 7% more LUT slices than the VHDL,
 5 while DSP and BRAM usage is identical. The MaxJ code used approximately half the number of
 6 lines of source code of VHDL.

7 3 CMS Level 1 Track Trigger Demonstrator

8 In order to maximally exploit the scientific potential of the LHC, the machine will be upgraded to
 9 provide a peak instantaneous luminosity of $5 - 7.5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ and a total integrated luminosity
 10 of 3000 fb^{-1} [5]. Construction will begin in 2024, and it is known as the High-Luminosity LHC
 11 (HL-LHC). In the 140-200 pile-up conditions expected, reconstructing tracks from the silicon
 12 tracker at Level-1, and exploiting them in the trigger, will be necessary in order to keep object
 13 energy thresholds as low as possible at the given trigger accept rate. A ‘ p_T module’ has been
 14 designed to reduce the data volume from the tracker to the trigger [6]. Charged particle tracks will
 15 be measured on each of two segmented silicon detector layers, spaced approximately 1 mm apart.
 16 The bend induced by the 3.8 T magnetic field causes a displacement between the layers. Pairs of
 17 measurements within a configurable window are clustered to form a ‘stub’. Only stubs are to be
 18 read out to the L1 track trigger, with a p_T threshold of 2 – 3 GeV. Approximately 98% of tracks are
 19 below 2 GeV and considered to be less useful for the trigger. A factor 10 reduction in data rate to
 20 the trigger is achieved by reading out stubs only.

21 The track trigger must reconstruct charged particle tracks with a latency of around $4 \mu\text{s}$. One
 22 proposal for a track trigger design is based on an FPGA implementation of a Hough Transform (HT)

[7]. The HT performs track finding in the plane transverse to the beam pipe, grouping stubs which form a viable trajectory in this 2-dimensional perspective. Following the HT, further steps are required to perform a fit to the stubs in 3 dimensions, filter the tracks to remove false associations of stubs, and remove duplicated tracks. A Kalman Filter [8] (KF) has been developed to carry out track fitting and simultaneous filtering. The Kalman Filter firmware is partly developed using MaxJ, and targets the MP7 board.

3.1 Kalman Filter

Kalman filters have been used extensively in tracking in HEP [9], and indeed on parallel architectures [10]. CMS uses a Kalman Filter at the High Level Trigger and for offline reconstruction. The filter begins with an estimate of the track parameters - provided by the preceding HT which found the track. Stubs are introduced to the KF iteratively, each time pulling the parameters by an amount dependent on the relative uncertainties of the measurement and the parameters, as seen in equations 3.1-3.9. Multiple stubs on the same detector layer are not considered to belong to the same track, and the candidate is split into independent trajectories. As a local method, the Kalman filter allows stubs to be tested for compatibility with a trajectory before updating the parameters. This is a useful feature for fitting track candidates found by the HT, which can contain stubs far from the true track in the z dimension. Another general feature of local track fitting methods is the need for smaller matrices than a global method for the same number of track parameters and measurements, which may yield an algorithm with lower resource usage in the FPGA.

The equations 3.1-3.9 update an estimate of the track parameters x_{k-1} and their covariance matrix $\mathbf{C}_{k-1}$ on layer $k-1$ to an estimate x_k and covariance matrix $\mathbf{C}_k$ on layer k . The matrix $\mathbf{F}_{k-1}$ projects the track parameters forwards with process noise $\mathbf{Q}_{k-1}$. $\mathbf{H}_k$ predicts the stub position on layer k , m_k is a stub measurement and $\mathbf{V}_k$ is the measurement uncertainty.

$$x_k^{k-1} = \mathbf{F}_{k-1} x_{k-1} \quad (3.1)$$

$$\mathbf{C}_k^{k-1} = \mathbf{F}_{k-1} \mathbf{C}_{k-1} \mathbf{F}_{k-1}^T + \mathbf{Q}_{k-1} \quad (3.2)$$

$$r_k^{k-1} = m_k - \mathbf{H}_k x_k^{k-1} \quad (3.3)$$

$$\mathbf{R}_k^{k-1} = \mathbf{V}_k + \mathbf{H}_k \mathbf{C}_k^{k-1} \mathbf{H}_k^T \quad (3.4)$$

$$\mathbf{K}_k = \mathbf{C}_k^{k-1} \mathbf{H}_k^T (\mathbf{R}_k^{k-1})^{-1} \quad (3.5)$$

$$x_k = x_k^{k-1} + \mathbf{K}_k r_k^{k-1} \quad (3.6)$$

$$\mathbf{C}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{C}_k^{k-1} \quad (3.7)$$

$$\chi_+^2 = r_k^{k-1T} (\mathbf{R}_k^{k-1})^{-1} r_k^{k-1} \quad (3.8)$$

$$\chi_k^2 = \chi_{k-1}^2 + \chi_+^2 \quad (3.9)$$

A simplified 4-parameter fit with linear equations is implemented, fitting $x = (1/2R, \phi_0, \tan \lambda, z_0)$ where R is the track radius of curvature – proportional to the p_T – ϕ_0 is the initial track angle in the x - y plane, $\tan \lambda$ is the track gradient in the r - z plane and z_0 is the vertex. The matrices in this parametrisation are a combination of 4×4 , 2×2 and 4×2 , and the matrix inversion is of a 2×2 .

Figure 3. The dataflow graph for element $(AB)_{ij}$ where A is a $(n \times 4)$ and B is a $(4 \times m)$ matrix. All multiplications are performed in parallel, and the sum of individual products uses a balanced adder tree.

3.1.1 MaxJ Implementation

An implementation of equations 3.1-3.9 was developed in MaxJ to process tracks found by the HT. The implementation uses exclusively fixed-point arithmetic, where each value has a fixed number of bits before and after the radix point. This allows the use of integer operations, which are cheaper in terms of FPGA resources and latency than floating-point operations. MaxCompiler contains many features to simplify the production of efficient fixed-point designs.

Let x and y be two signed numbers, with n_x and n_y bits respectively. The result of $x \times y$ will at most use $n_x + n_y - 1$ bits to represent all possible outcomes. In the Xilinx Virtex-7 FPGA targeted for this application, the DSPs used to perform multiplications have inputs for one 18-bit and one 25-bit signed number. Multiple DSPs can be used if either multiplicand is larger than the single DSP input size.

It is evident that if the product $x \times y$ is to be used in another multiplication - as occurs frequently in equations 3.1-3.9 - then the number of bits used to encode $x \times y$ must be kept to no more than 18 or 25 bits in order to use only 1 DSP for the subsequent multiplication and keep the overall resource usage small. The developer can select from a collection of strategies to allow the compiler to set the bit width and radix point location for the output. After setting the bit size to be either 18 or 25, for minimal DSP usage, the tool is able to set the radix point given a requirement to avoid either underflow or overflow. Alternatively the developer is able to set the point manually. This feature was used extensively in the Kalman Filter firmware, achieving a DSP usage of one per multiplication operation.

A matrix multiplication is given by $(\mathbf{AB})_{ij} = \sum_k A_{ik} B_{kj}$. Since the matrices in this Kalman Filter are small, each $A_{ik} \times B_{kj}$ is executed in parallel, with the summation performed as a balanced adder tree on the products. Figure 3 shows the dataflow graph for a matrix multiplication for one element of a $(n \times 4) \times (4 \times m)$ product. Furthermore, matrix components which are constants known at compile time are optimised by the compiler, as described previously. The division required

Figure 4. The absolute relative difference between the output of fixed-point firmware (fw) and floating-point C++ software (sw) implementations of the Kalman Filter, for each update iteration when using simulated $\sqrt{s}$ collisions with 140 pile-up. The difference between the two implementations is less than 0.5% for most iterations.

1 to find the matrix inverse is performed using an efficient, custom algorithm requiring 1 BRAM and
2 1 DSP. The implementation of a single Kalman Filter (equations 3.1-3.9) uses approximately 1%
3 each of the LUTs, DSPs, and BRAMs of a Xilinx Virtex-7 XC7VX690T FPGA on the MP7 board,
4 and is capable of running at the desired 240 MHz clock frequency, with a single iteration latency of
5 45 clock cycles, or 187.5 ns.

6 3.1.2 Simulation

7 MaxCompiler contains features to facilitate communication between the firmware and other CPU
8 applications - whether in hardware, or simulation. In developing the Kalman Filter, the MaxCom-
9 piler simulation was found to be an extremely useful feature. The tool compiles the firmware into a
10 model which uses the data-flow graph and libraries supporting the use of arbitrary bit width maths
11 operations. This model takes the form of a '.max' file, which can be most directly interfaced with
12 a C/C++ application, but can also be used with other languages such as Python and MATLAB. A
13 function call passes formatted data to the firmware (whether in simulation or hardware) and the
14 output can be processed on the CPU.

15 For the Kalman Filter this was used to test the firmware on Monte-Carlo stubs using the CMS
16 experiment software, CMSSW, and a simulation framework for studying the Hough Transform
17 performance. This direct simulation from within a C++ application facilitated comparisons between
18 the Kalman Filter firmware and a floating point C++ implementation of the algorithm on a per-
19 iteration basis, as seen in figure 4 for the $1/2R$ and ϕ_0 track parameters. The majority of update
20 operations are seen to match between the two implementations to better than 0.5%.

21 In addition to comparisons of the firmware output with a software model, debug information
22 was used to tune and debug the MaxJ. The available debug data includes the collection of results

1 of intermediate calculations within the firmware, and numerical exception information, making it
2 easy to find type underflows and overflows from incorrect truncations.

3 **3.2 Control Flow**

4 Stubs and states are passed into the KF by a module, written in VHDL, which stores the objects
5 in a RAM and a FIFO respectively. The control flow module maintains an address book for stubs,
6 and loops over the stubs in a candidate in order of layer. Due to the pipelined design of the filtering
7 equations, it is possible to pass new data into the filter at every clock cycle. This allows a single KF
8 processor to filter multiple independent track candidates simultaneously. This design demonstrates
9 the possibility of developing an application in a combination of VHDL and MaxJ. An advantage of
10 this approach is the decoupling of the data-flow and control-flow, whereby the high level language
11 is used to describe the maths operations, and the HDL is used to move data within the FPGA. A
12 configuration including 36 independent KF processors has been placed and routed in the target
13 FPGA and is capable of operation at 240 MHz.

14 **4 Summary**

15 MaxJ, a high level synthesis language, has been explored as a design tool for trigger applications
16 in High Energy Physics. By implementing existing algorithms from the CMS calorimeter trigger,
17 written originally in VHDL, in MaxJ, we have shown that MaxCompiler can produce identical
18 results, and with only a small difference in resource usage. The MaxJ code for the calorimeter
19 trigger application requires approximately half the number of code lines of the VHDL. The smaller
20 code-base, together with the higher level of abstraction offered by the MaxJ language, suggest the
21 possibility of developing more maintainable, and more widely understood, trigger algorithms.

22 We have then demonstrated the capabilities of the compiler by developing a complex track
23 fitting algorithm for the CMS Time Multiplexed Track Trigger demonstrator. Features of the tool
24 which assist the development of such an application have been highlighted, including the fixed-point
25 optimisations and the simulation of the firmware interfaced with a C++ application. The resulting
26 firmware achieves a good matching with a C++ floating-point implementation, and each Kalman
27 Filter utilises just 1% of the resources of a Xilinx Virtex-7 XC7VX690T, allowing 36 to be used in
28 parallel at 240 MHz. The Kalman Filter must now be integrated into the Time Multiplexed Track
29 Trigger demonstrator system.

30 **Acknowledgments**

31 We gratefully acknowledge the UK Science and Technologies Facilities Council and Maxeler
32 Technologies Ltd. for their support.

33 **References**

- 34 [1] Maxeler Technologies, *MaxCompiler (White Paper)* (Feb 2011), Available:
35 <https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf>
36 [2] The CMS Collaboration, *The CMS Experiment at the CERN LHC*, *JINST* **3** (2008)

- 1 [3] B. Kreis et al., *Run 2 Upgrades to the CMS Level-1 Calorimeter Trigger*, *JINST* **11** (2016)
- 2 [4] K. Compton, A. Rose et al., *The MP7 and CTP-6: Multi-Hundred Gbps Processing Boards for*
3 *Calorimeter Trigger Upgrades at CMS*, *JINST* **7** C12024 DOI: 10.1088/1748-0221/7/12/C12024
- 4 [5] G. Apollinari, I. Béjar Alonso, O. Brüning, M. Lamont, L. Rossi, *High-Luminosity Large Hadron*
5 *Collider (HL-LHC): Preliminary Design Report*, DOI: 10.5170/CERN-2015-005
- 6 [6] J. Jones, G. Hall, C. Foudas, A. Rose, *A Pixel Detector for Level-1 Triggering at SLHC, 11th LECC*
7 *Workshop* (2005)
- 8 [7] C. Amstutz et al., *An FPGA-Based Track Finder for the L1 Trigger of the CMS Experiment at the High*
9 *Luminosity LHC, Real Time Conference (RT), 2016 IEEE-NPSS*, 6-10 June 2016, DOI:
10 10.1109/RTC.2016.7543102
- 11 [8] R. E. Kálmán, *A New Approach to Linear Filtering and Prediction Problems*, *J BASIC ENG-T ASME*,
12 **82** Series D (1960), 35-45
- 13 [9] R. Frühwirth, *Application of Kalman Filtering to Track and Vertex Fitting*, *NIM* **A262** (1987)
14 444-450, DOI:10.1016/0168-9002(87)90887-4
- 15 [10] G. Cerati et al., *Kalman Filter Tracking on Parallel Architectures*, *J. Phys Conf. Ser.* **664** (2015)