Imperial College London

Fast FPGA prototyping with Software Development Kit for FPGA (SDK4FPGA)

Andrea Suardi
cas.ee.ic.ac.uk/projects/SDK4FPGA

This research has been supported by
EPSRC Impact Acceleration grant number EP/K503733/1

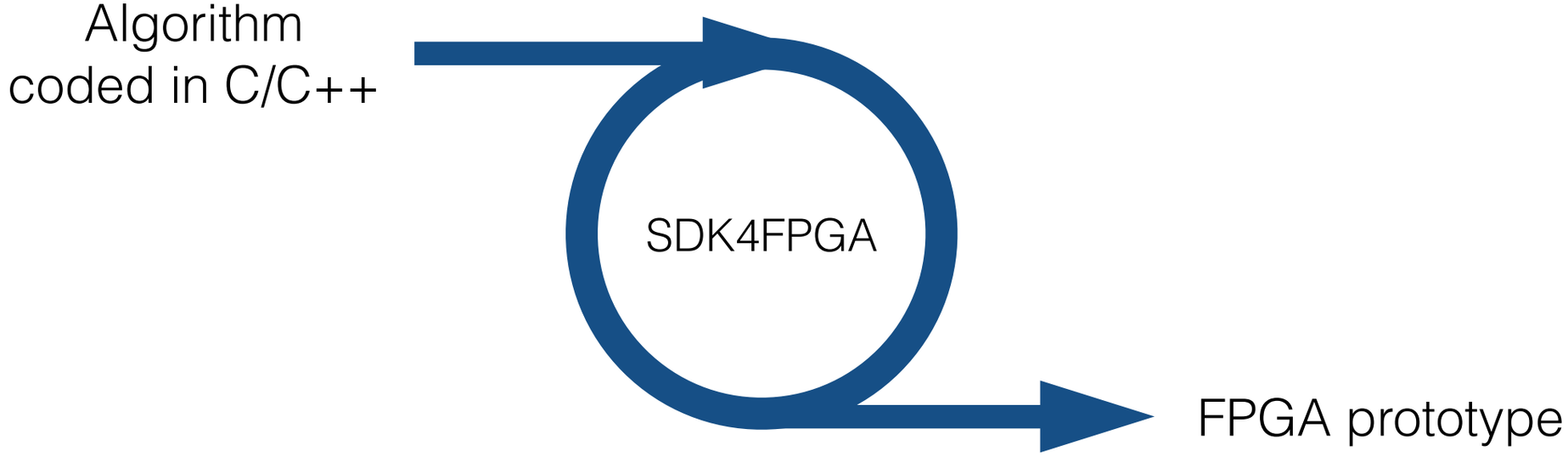
Outline

- What is SDK4FPGA ?
- Why SDK4FPGA for embedded optimisation?
- How does SDK4FPGA work ?

(Case study: Fast Gradient for real-time audio processing)

1. Algorithm coding
2. Verification (off-line simulation)
3. FPGA prototype

What is SDK4FPGA ?

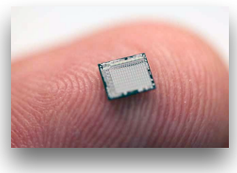


- Open Source framework
- Automated design flow
- Customisable templates and example designs

Why SDK4FPGA for embedded optimisation?

Pros:

- fast FPGA prototype **[< 1 day]**
  - low power consumption **[<1W]**
  - low cost **[<10$]**
  - applications with fast dynamics **[~ms-µs]**
  - small packaging

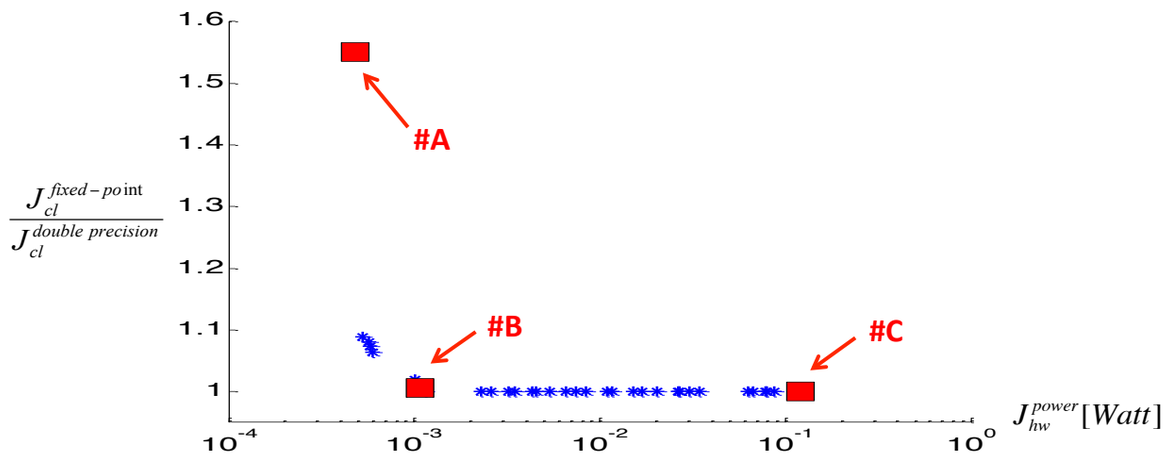


- easy algorithm numerical validation **[floating-point, fixed-point]**
- no FPGA knowledge required

Cons:

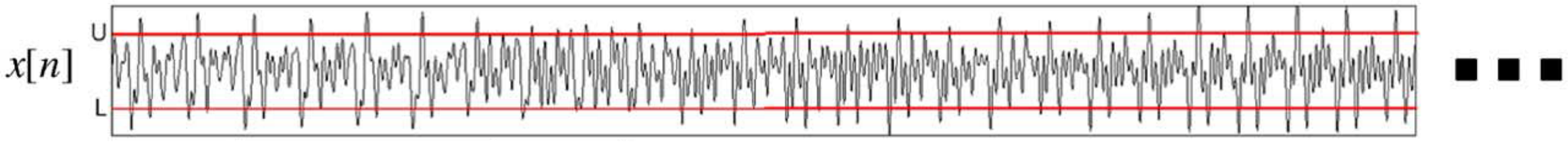
- algorithm already C/C++ coded and verified
- not Matlab to FPGA coding support
- think parallel / small memory

automated circuit design optimisation support



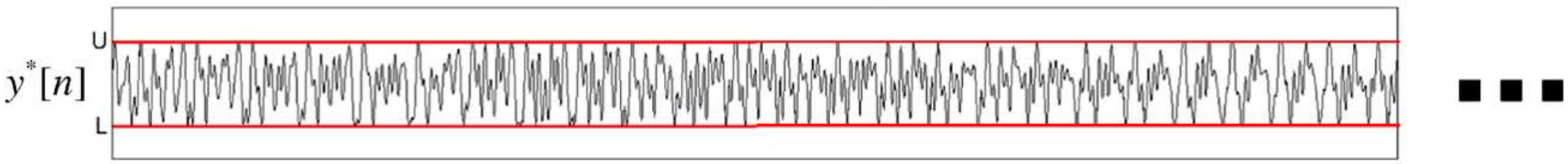
Fast Gradient for real-time audio processing (CLIP algorithm)

Real-time perception-based clipping of audio signals using convex optimisation
B. Defraene, T. van Waterschoot, H.J. Ferreau, M. Diehl, and M. Moonen
IEEE Transactions on, Audio, Speech, and Language Processing



Fast Gradient Method

Configuration parameters



Fast Gradient for real-time audio processing (CLIP algorithm)

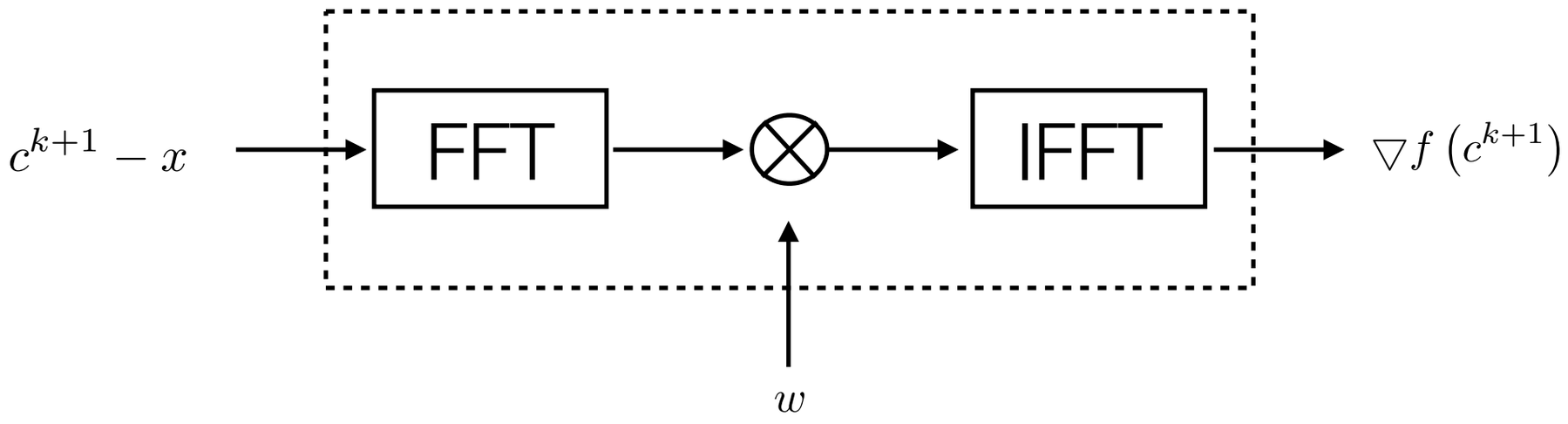
Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\boldsymbol{\delta} = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T \in \mathbb{R}^{K_{\max}}$ L, U, C^{-1}
Output $\mathbf{y}^* \in \mathbb{R}^N$

$\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
$\nabla f(\mathbf{c}^0) = \mathbf{0}$
$k = 0$
while $k < K_{\max}$ **do**
  $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1} \nabla f(\mathbf{c}^k)$
  $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
  $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k (\mathbf{y}^{k+1} - \mathbf{y}^k)$
  $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w}) \mathbf{D} (\mathbf{c}^{k+1} - \mathbf{x})$
  $k = k + 1$
end while
$\mathbf{y}^* = \mathbf{y}^k$

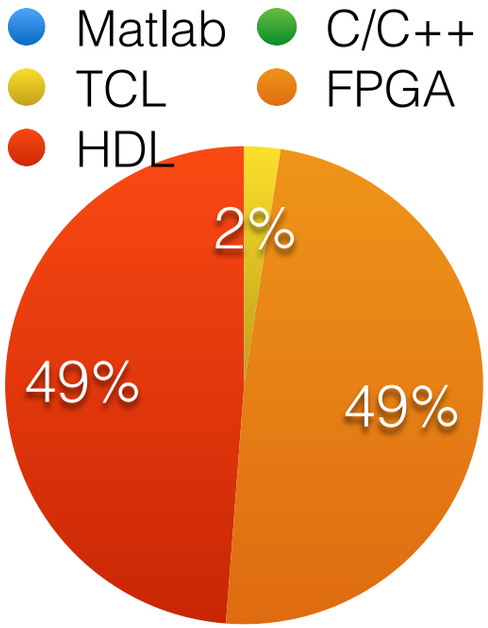
Fast Gradient for real-time audio processing (CLIP algorithm)

Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\boldsymbol{\delta} = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T \in \mathbb{R}^{K_{\max}}$ L, U, C^{-1}
Output $\mathbf{y}^* \in \mathbb{R}^N$

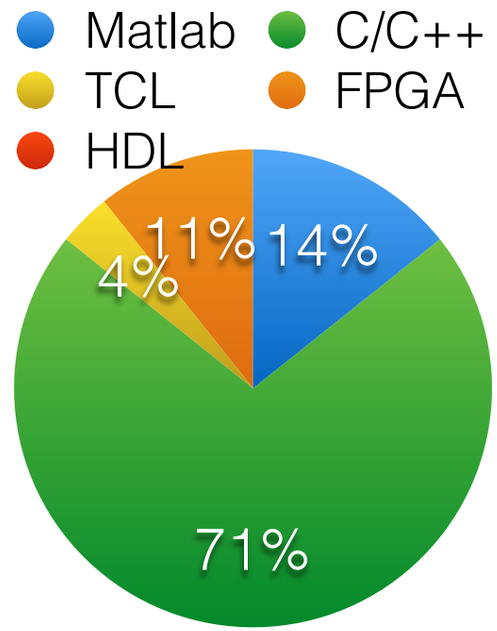
$\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
$\nabla f(\mathbf{c}^0) = \mathbf{0}$
$k = 0$
while $k < K_{\max}$ **do**
  $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1} \nabla f(\mathbf{c}^k)$
  $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
  $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k(\mathbf{y}^{k+1} - \mathbf{y}^k)$
  $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w}) \mathbf{D}(\mathbf{c}^{k+1} - \mathbf{x})$
  $k = k + 1$
end while
$\mathbf{y}^* = \mathbf{y}^k$



1. Algorithm coding



conventional hand-coded HDL approach



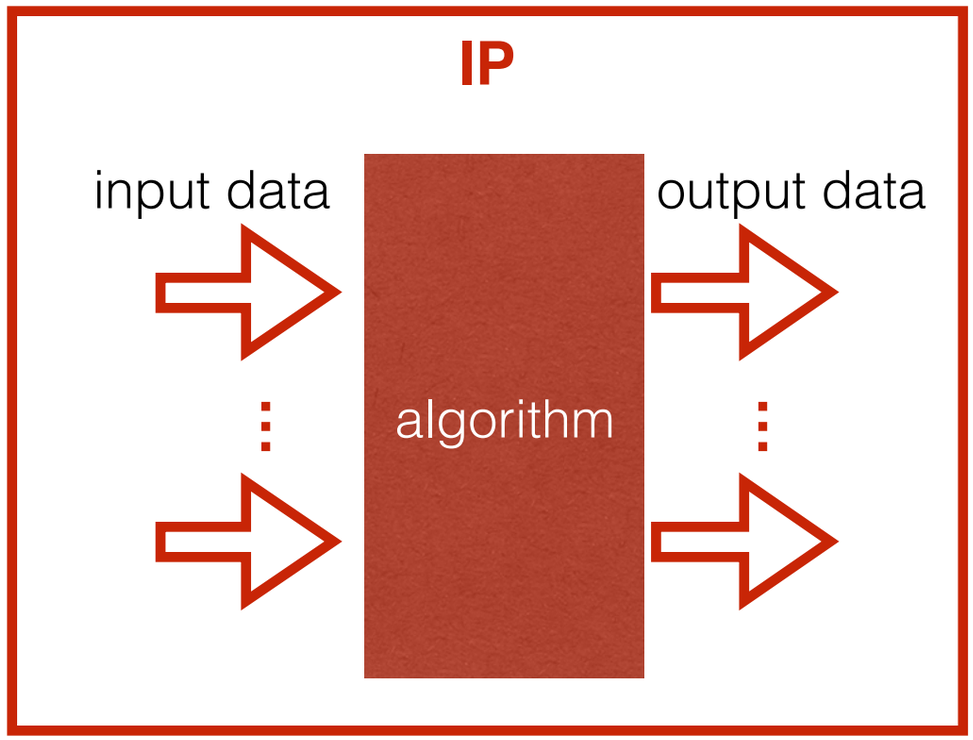
nowadays High Level Synthesis approach

1. Algorithm coding

radar design 1024 x 64 QRD floating point	conventional hand-coded HDL approach	nowadays High Level Synthesis approach
Design language	VDHL/Verilog	**C**
Design Time (weeks)	12	**1**
Latency (ms)	37	**21**
Memory (RAMB36E1)	273	**138**
Registers	29826	**14263**
Logic (LUTs)	28152	**24257**

www.xilinx.com

1. Algorithm coding



- **User:**
  - defines input/output data:
    - scalar
    - vector of any size
  - defines data representation:
    - floating-point single precision
    - any fixed-point up to 32 bits word length
  - *codes algorithm in C/C++*

- **SDK4FPGA**:
  - provides a customised function template
  - calls Xilinx Vivado HLS to build the circuit

1. Algorithm coding

Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\delta = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T \in$
Output $\mathbf{y}^* \in \mathbb{R}^N$

$\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
$\nabla f(\mathbf{c}^0) = \mathbf{0}$
$k = 0$
while $k < K_{\max}$ **do**
  $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1} \nabla f(\mathbf{c}^k)$
  $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
  $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k(\mathbf{y}^{k+1} - \mathbf{y}^k)$
  $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w}) \mathbf{D}(\mathbf{c}^{k+1} - \mathbf{x})$
  $k = k + 1$
end while
$\mathbf{y}^* = \mathbf{y}^k$

```
#define NUMBER_ITERATIONS 30
#define INTEGER_LENGTH 4
#define FRACTION_LENGTH 8

#define N 512

typedef ap_fixed< INTEGER_LENGTH+FRACTION_LENGTH,
 INTEGER_LENGTH,AP_TRN, AP_SAT> data_t;

void clip(
          data_t x[N],
          data_t w[N],
          data_t bmin[N],
          data_t bmax[N],
          data_t delta[Kmax],
          data_t lipschitz,
          data_t y_out[N])
{

//variables
data_t Grad[N];
data_t Grad_lipschitz[N];
data_t new_Grad[N];
data_t y_tilde[N];
data_t y_new[N];
data_t y[N];
data_t y_delta[N];
data_t y_delta_delta[N];
data_t c_new[N];
data_t c[N];

int k,i;
```

Memory

1. Algorithm coding

Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\boldsymbol{\delta} = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T \in \mathbb{R}^{K_{\max}}$ L, U, C^{-1}
Output $\mathbf{y}^* \in \mathbb{R}^N$

$\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
$\nabla f(\mathbf{c}^0) = 0$
$k = 0$
while $k < K_{\max}$ **do**
  $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1} \nabla f(\mathbf{c}^k)$
  $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
  $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k(\mathbf{y}^{k+1} - \mathbf{y}^k)$
  $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w}) \mathbf{D}(\mathbf{c}^{k+1} - \mathbf{x})$
  $k = k + 1$
end while
$\mathbf{y}^* = \mathbf{y}^k$

```
//initialization
initialization_loop: for (i=0; i< N; i++)
{
    Grad[i]=0;
    c[i]=x[i]
    y[i]=x[i];
}
```

Executed in N steps

1. Algorithm coding

Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\boldsymbol{\delta} = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T \in \mathbb{R}^{K_{\max}}$ L, U,
Output $\mathbf{y}^* \in \mathbb{R}^N$

$\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
$\nabla f(\mathbf{c}^0) = \mathbf{0}$
$k = 0$
while $k < K_{\max}$ **do**
  $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1} \nabla f(\mathbf{c}^k)$
  $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
  $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k(\mathbf{y}^{k+1} - \mathbf{y}^k)$
  $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w}) \mathbf{D}(\mathbf{c}^{k+1} - \mathbf{x})$
  $k = k + 1$
end while
$\mathbf{y}^* = \mathbf{y}^k$

```
// Fast Gradient iterations loop
FG_loop:for (int k=0; k< NUMBER_ITERATIONS; k++)

    //Iteration
    inner_loop_row:    for(i = 0; i < N; i++)
    {
        //Gradient * Lipschitz
        Grad_lipschitz[i] = Grad[i] * lipschitz;

        //unconstrained update
        y_tilde[i]=c[i]-Grad_lipschitz[i];

        //projection
        if (y_tilde[i]>bmax[i])
            y_new[i]=bmax[i];
        else if (y_tilde[i]<bmin[i])
            y_new[i]=bmin[i];
        else
            y_new[i]=y_tilde[i];

        //update c
        y_delta[i]=y_new[i]-y[i];
        y_delta_delta[i]=delta[k] * y_delta[i];
        c_new[i]=y_new[i]+y_delta_delta[i];

        to_fft[i]=c_new[i]-x[i];
    }

    // FFT
    hls::fft(to_fft, fft_out);

    //apply weights
    w_loop: for (i=0; i< N; i++)
    {
        to_ifft[i].real()=fft_out[i].real()*w[i];
        to_ifft[i].imag()=fft_out[i].imag()*w[i];
    }

    // IFFT
    hls::ifft(to_ifft, new_Grad);

    //update variables
    update_loop: for (i=0; i< N; i++)
    {
        Grad[i]=new_Grad[i];
        c[i]=c_new[i]
        y[i]=y_new[i];
    }
}
```

1. Algorithm coding

Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\boldsymbol{\delta} = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T$ ϵ
Output $\mathbf{y}^* \in \mathbb{R}^N$

1: $\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
2: $\nabla f(\mathbf{c}^0) = \mathbf{0}$
3: $k = 0$
4: **while** $k < K_{\max}$ **do**
5: $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1}\nabla f(\mathbf{c}^k)$
6: $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
7: $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k(\mathbf{y}^{k+1} - \mathbf{y}^k)$
8: $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w})\mathbf{D}(\mathbf{c}^{k+1} - \mathbf{x})$
9: $k = k + 1$
10: **end while**
11: $\mathbf{y}^* = \mathbf{y}^k$

Pipeline: Executed in N+7 steps

builtin function

```
// Fast Gradient iterations loop
FG_loop:for (int k=0; k< NUMBER_ITERATIONS; k++)

    //Iteration
    inner_loop_row:    for(i = 0; i < N; i++)
    {
        //Gradient * Lipschitz
        Grad_lipschitz[i] = Grad[i] * lipschitz;

        //unconstrained update
        y_tilde[i]=c[i]-Grad_lipschitz[i];

        //projection
        if (y_tilde[i]>bmax[i])
            y_new[i]=bmax[i];
        else if (y_tilde[i]<bmin[i])
            y_new[i]=bmin[i];
        else
            y_new[i]=y_tilde[i];

        //update c
        y_delta[i]=y_new[i]-y[i];
        y_delta_delta[i]=delta[k] * y_delta[i];
        c_new[i]=y_new[i]+y_delta_delta[i];

        to_fft[i]=c_new[i]-x[i];
    }

    // FFT
    hls::fft(to_fft, fft_out);

    //apply weights
    w_loop: for (i=0; i< N; i++)
    {
        to_ifft[i].real()=fft_out[i].real()*w[i];
        to_ifft[i].imag()=fft_out[i].imag()*w[i];
    }

    // IFFT
    hls::ifft(to_ifft, new_Grad);

    //update variables
    update_loop: for (i=0; i< N; i++)
    {
        Grad[i]=new_Grad[i];
        c[i]=c_new[i]
        y[i]=y_new[i];
    }
}
```

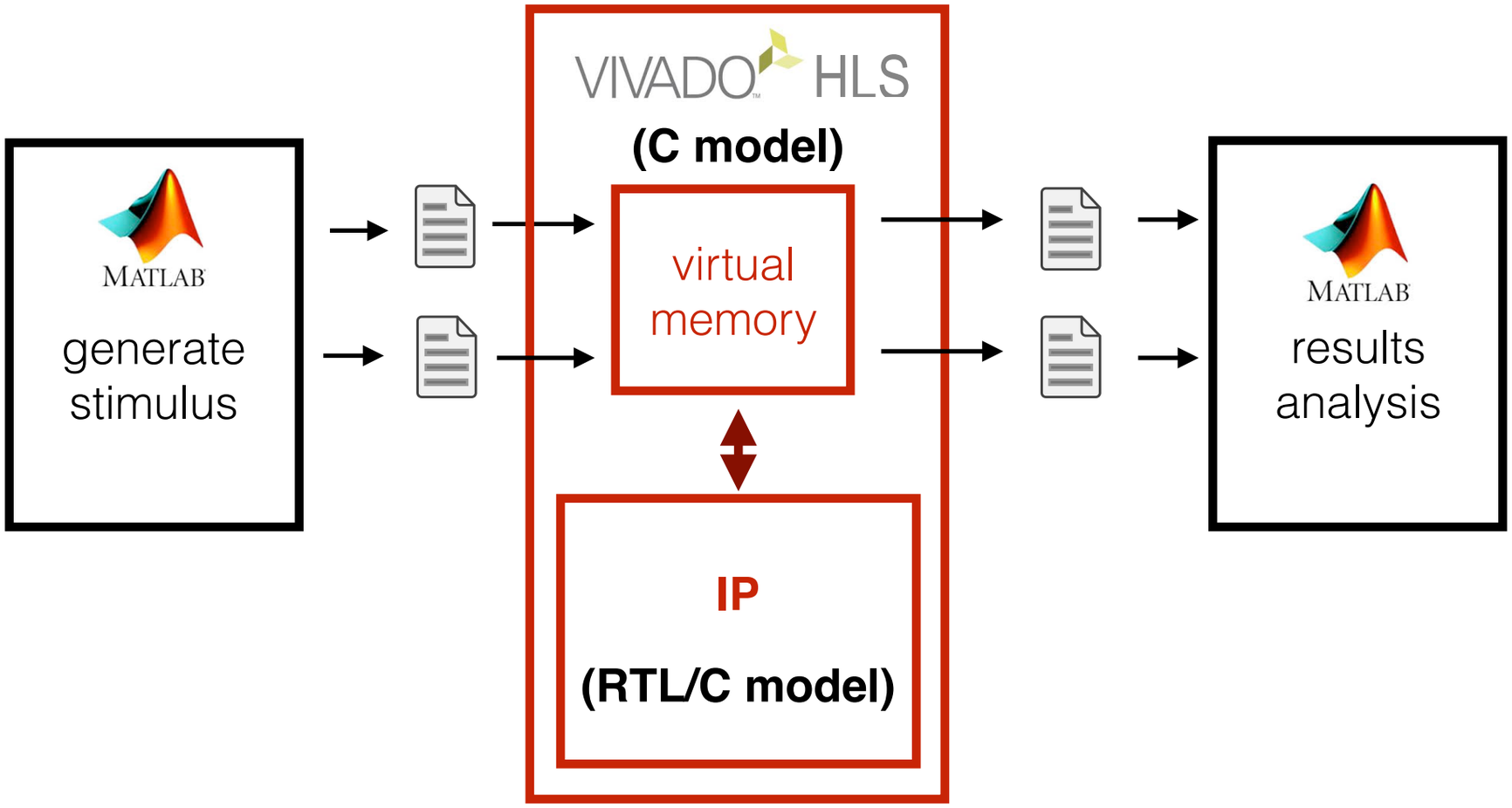
1. Algorithm coding

Input $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{w} \in \mathbb{R}^N$, $K_{\max}$, $\boldsymbol{\delta} = [\delta^0 \delta^1 \dots \delta^{K_{\max}-1}]^T \in \mathbb{R}^{K_{\max}}$ L, U, C^{-1}
Output $\mathbf{y}^* \in \mathbb{R}^N$

$\mathbf{y}^0 = \mathbf{c}^0 = \mathbf{x}$
$\nabla f(\mathbf{c}^0) = \mathbf{0}$
$k = 0$
while $k < K_{\max}$ **do**
  $\tilde{\mathbf{y}}^{k+1} = \mathbf{c}^k - C^{-1} \nabla f(\mathbf{c}^k)$
  $\mathbf{y}^{k+1} = \Pi_Q(\tilde{\mathbf{y}}^{k+1})$
  $\mathbf{c}^{k+1} = \mathbf{y}^{k+1} + \delta^k(\mathbf{y}^{k+1} - \mathbf{y}^k)$
  $\nabla f(\mathbf{c}^{k+1}) = \mathbf{D}^H \mathrm{diag}(\mathbf{w}) \mathbf{D}(\mathbf{c}^{k+1} - \mathbf{x})$
  $k = k + 1$
end while
$\mathbf{y}^* = \mathbf{y}^k$

```
//update output
update_output_loop: for (i=0; i< N; i++)
{
    y_out[i]=y[i];
}
```

2. Verification (off-line simulation)

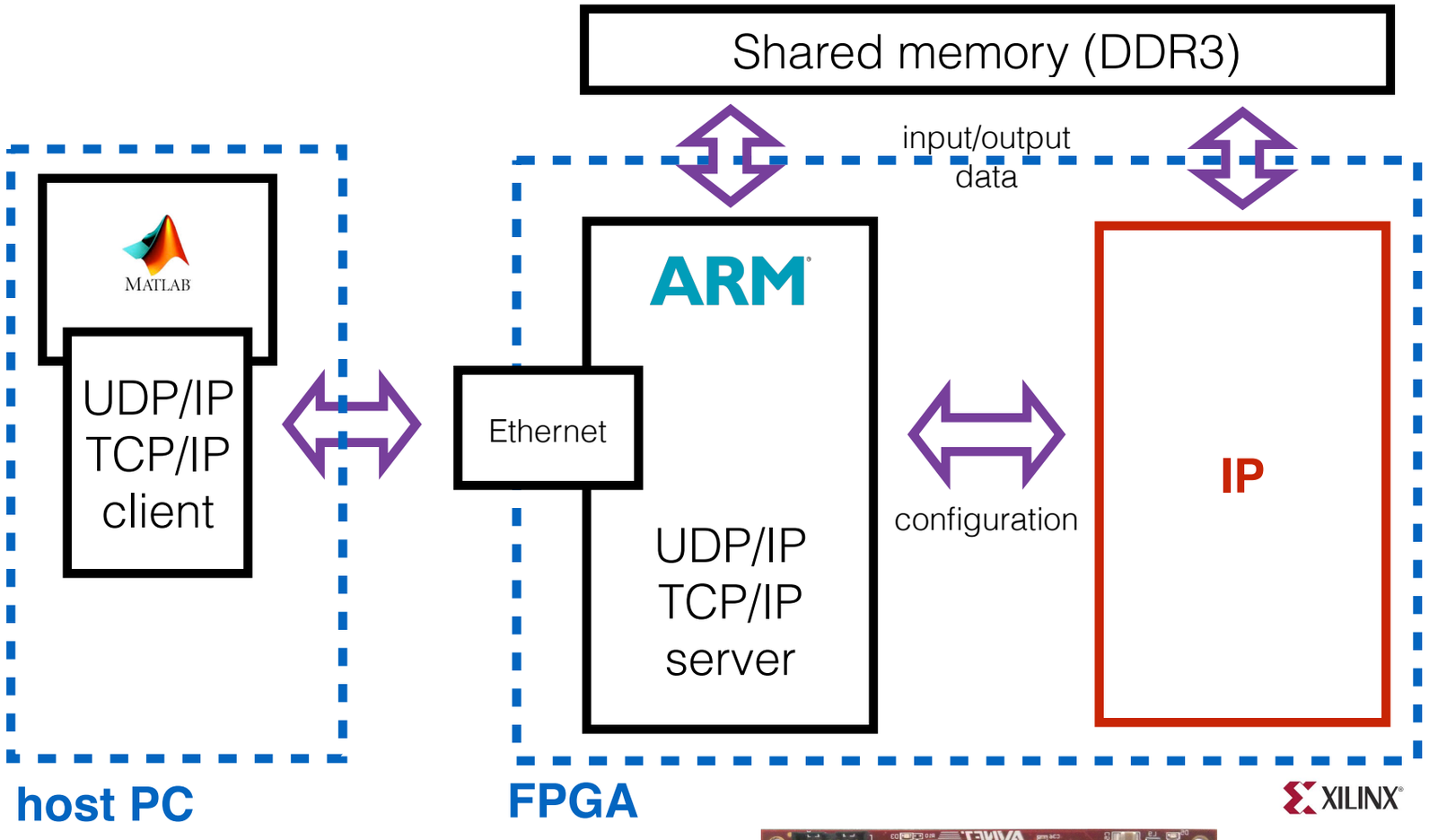


- **User:**
  - provides stimulus and analyses results from Matlab
  - defines computing precision

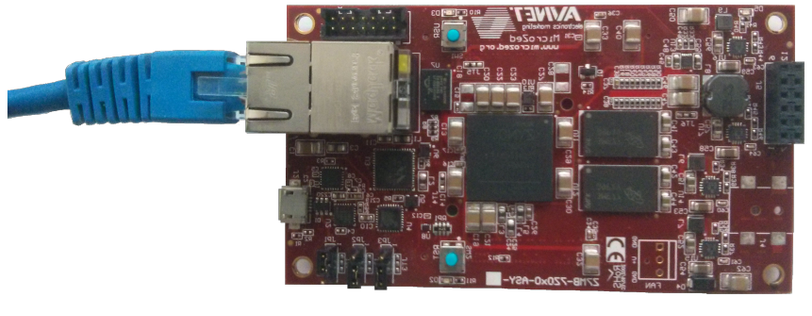
- **SDK4FPGA**:
  - handles the simulation interfacing Matlab with Xilinx Vivado HLS
  - reports circuit latency (delay) and resources (silicon Area)

3. FPGA prototype

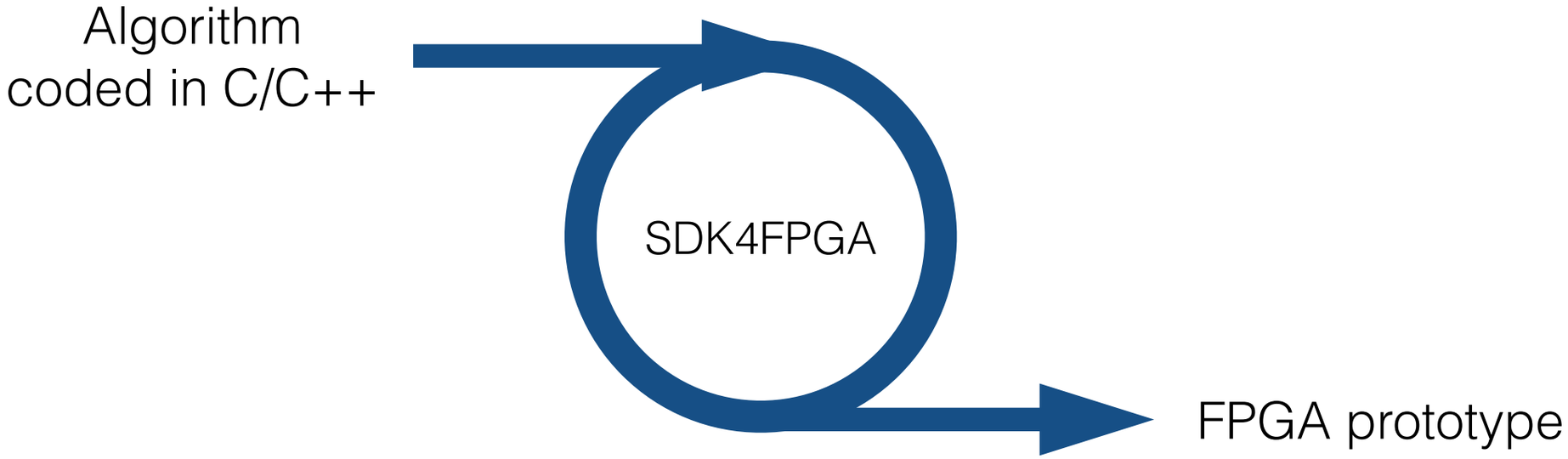
- **User:**
  - provides stimulus and analyses results with a Matlab API
  - defines target Evaluation Board
  - selects host PC interface (UDP/TCP)



- **SDK4FPGA**:
  - builds the FPGA circuit calling Xilinx Vivado
  - handle communication between host PC and FPGA



cas.ee.ic.ac.uk/projects/SDK4FPGA
Andrea Suardi [a.suardi@imperial.ac.uk]



This research has been supported by
EPSRC Impact Acceleration grant number EP/K503733/1