

# Study of combining GPU/FPGA accelerators for High-Performance Computing



Bruno Da Silva, An Braeken, Erik D'Hollander,  
Abdellah Touhafi, Jan G. Cornelis and Jan Lemeire

# Overview

1. HPC Desktop: CPUs + GPUs + FPGAs
2. Roofline performance of GPU/FPGA
3. Comparing GPU/FPGA for an image processing algorithm
4. GPU/FPGA collaboration: Pedestrian recognition
5. Conclusions

# 1. High Performance Desktop

Combining CPUs + GPUs + FPGAs

# HPC Desktop: CPU+FPGA+GPU

- Architecture:
  - CPU: Xeon E5506
  - GPU: Tesla C2050
  - FPGA: Pico Ex500 board with 2x Virtex6-LX240
- Toolchain:
  - Languages: C/C++ and OpenCL
  - High-Level Synthesis tools: ROCCC and VivadoHLS.
  - APIs: Nvidia Libraries(GPU) and Pico Computing framework (FPGA)

# HPC Desktop: CPU+FPGA+GPU

Tesla C2050

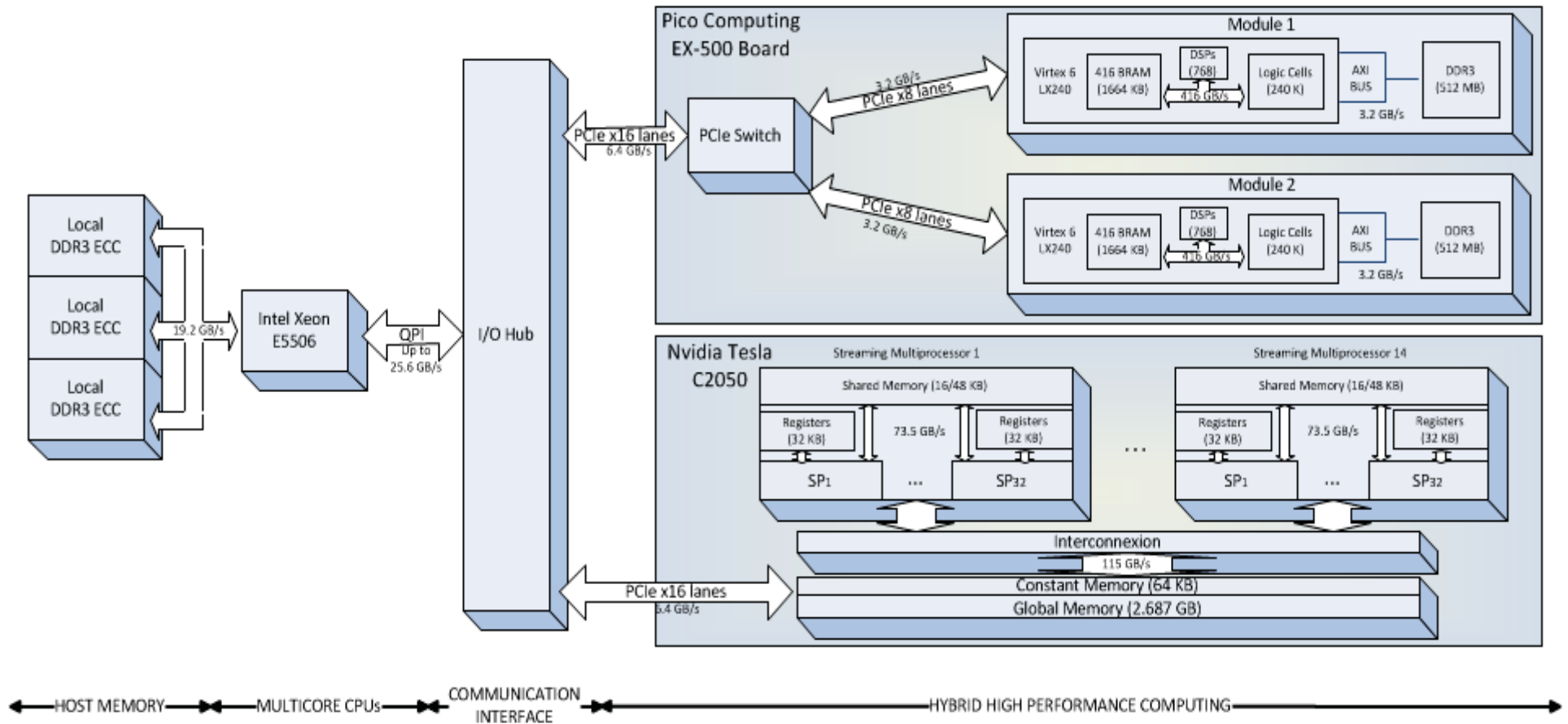


2x  
M501 Virtex6

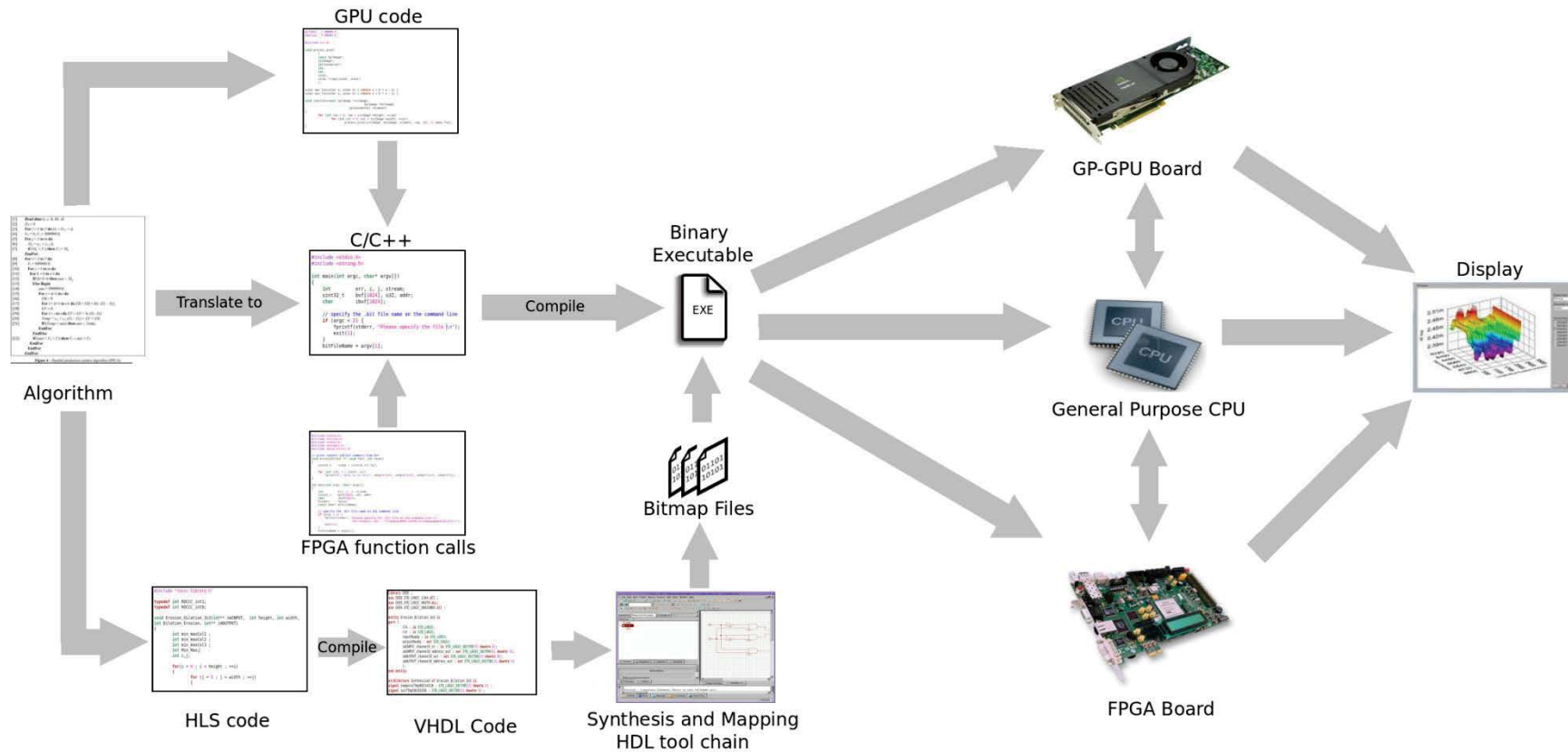


Pico EX-500

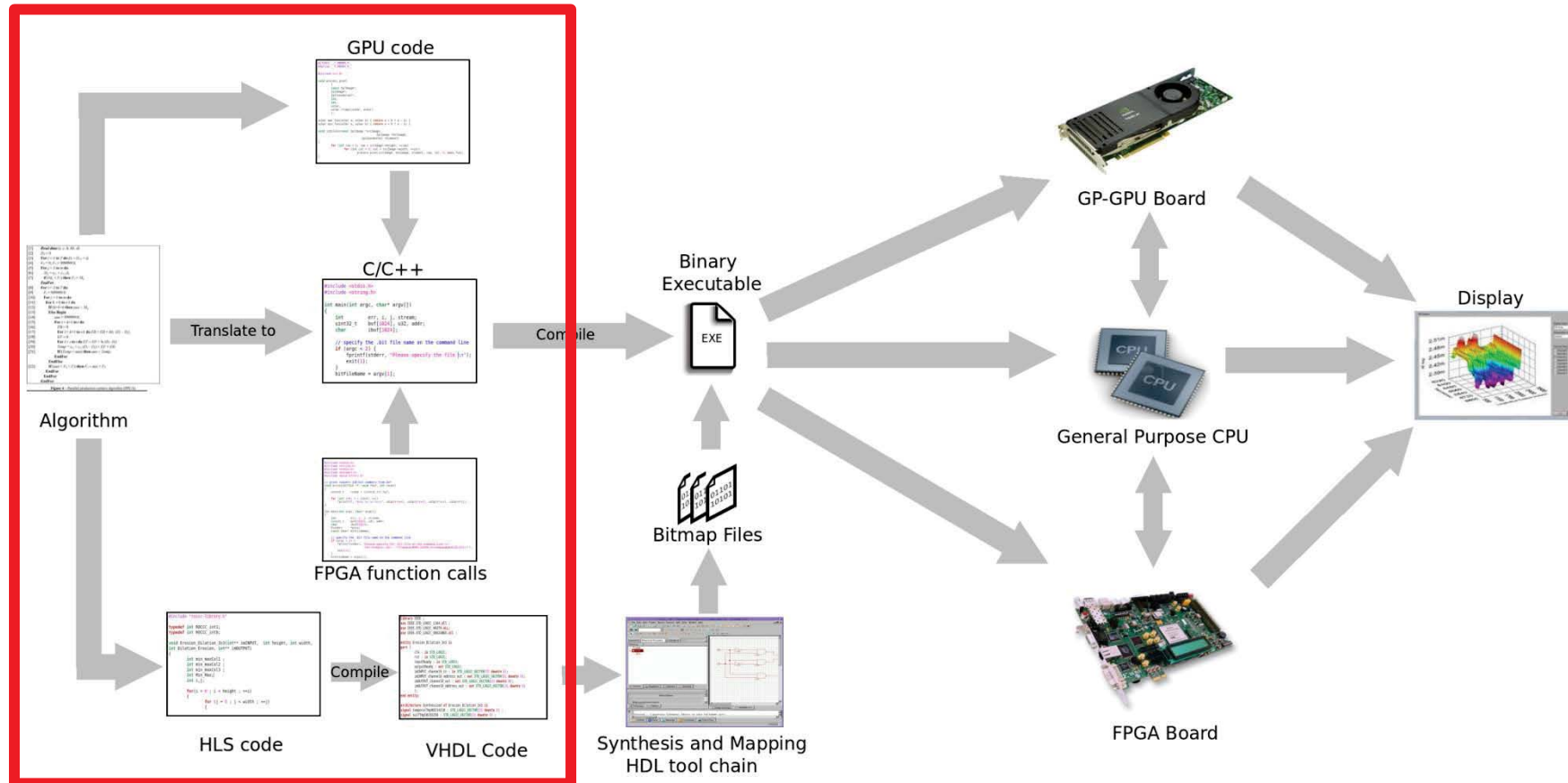
# CPU/FPGA/GPU Heterogeneous Architecture



# Toolchain combining GPU/FPGA/CPU

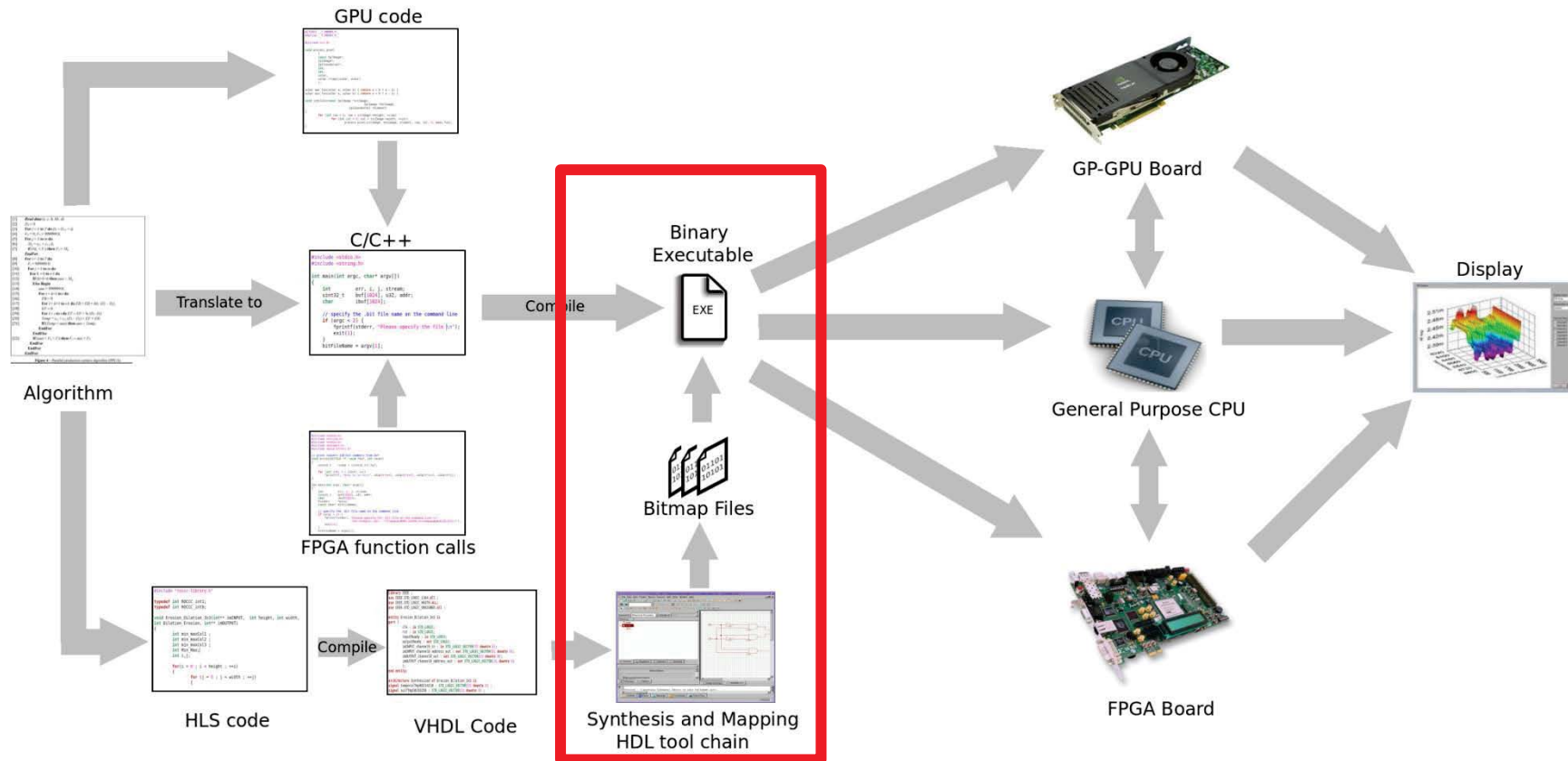


# Toolchain combining GPU/FPGA/CPU

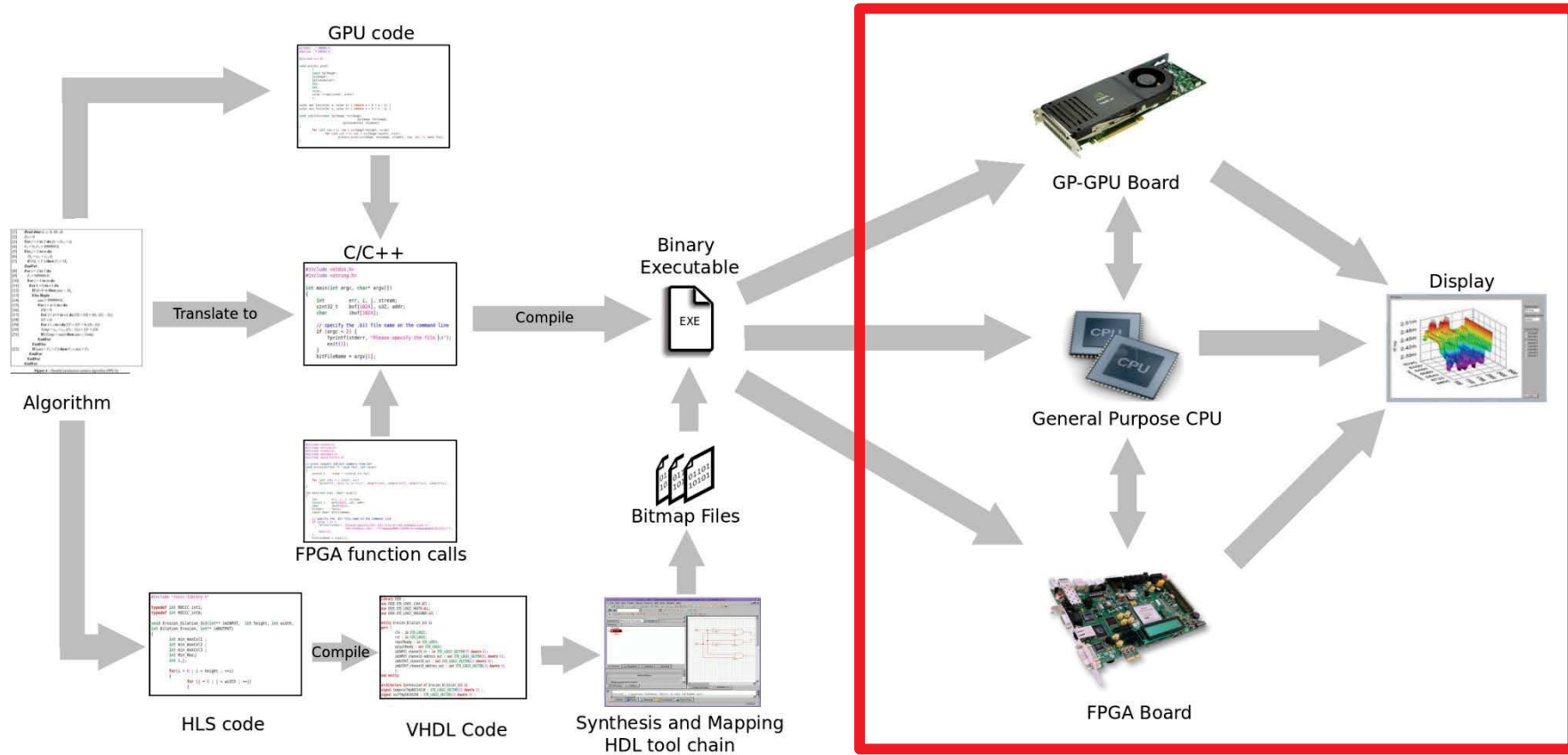




# Toolchain combining GPU/FPGA/CPU



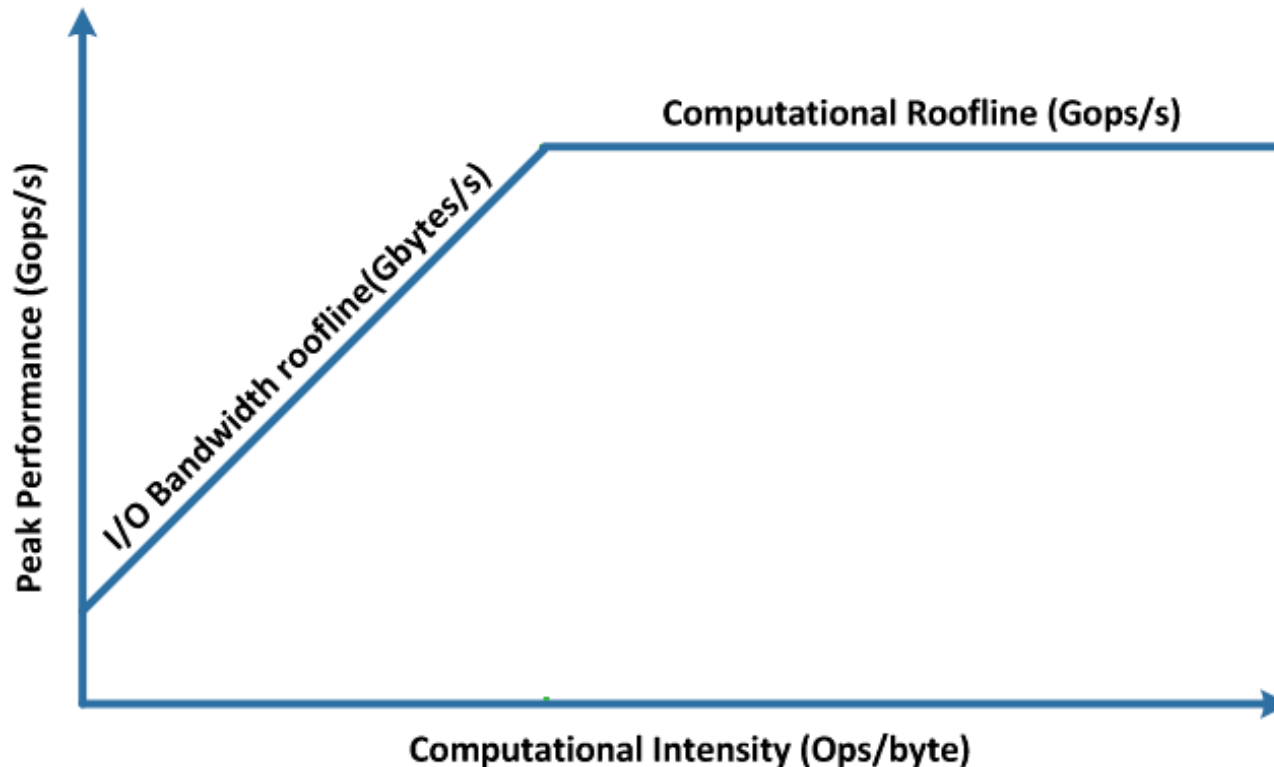
# Toolchain combining GPU/FPGA/CPU



# 2. Roofline Performance of GPU/FPGAs

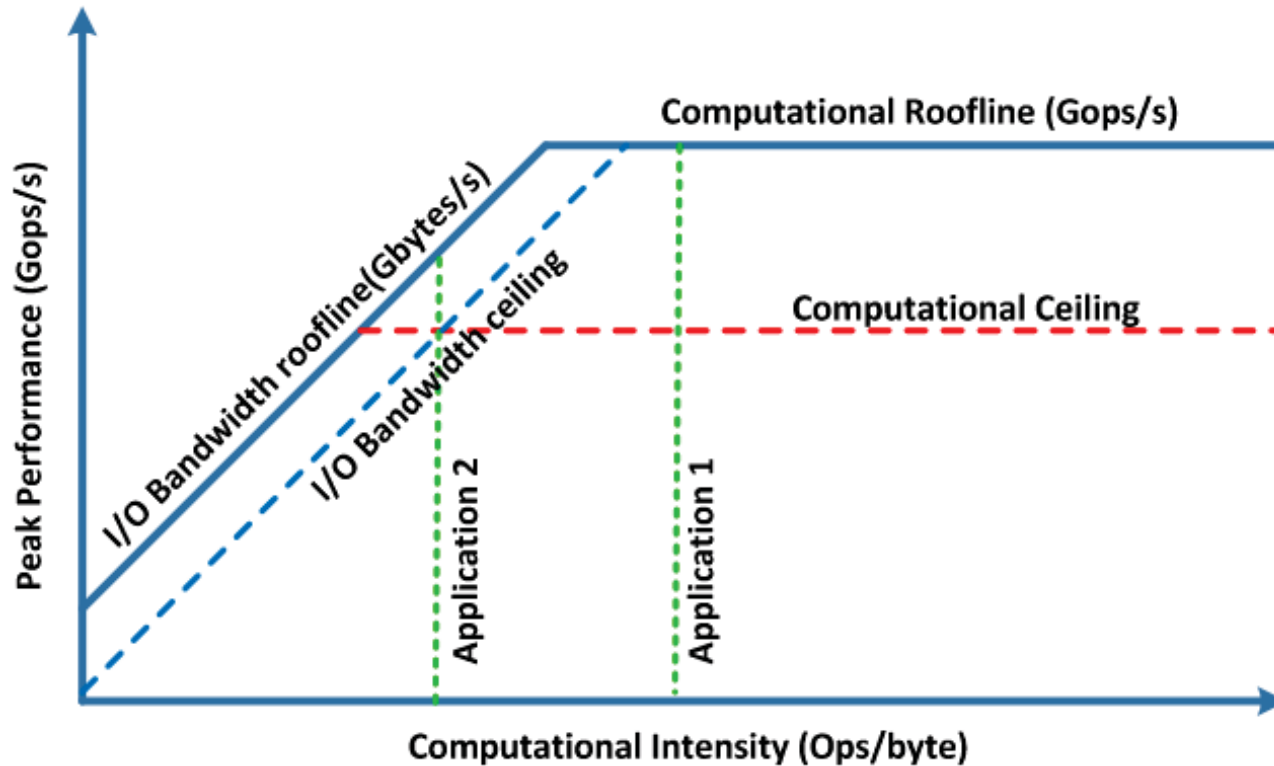
Adapting the roofline model for hardware accelerators

# Roofline model:



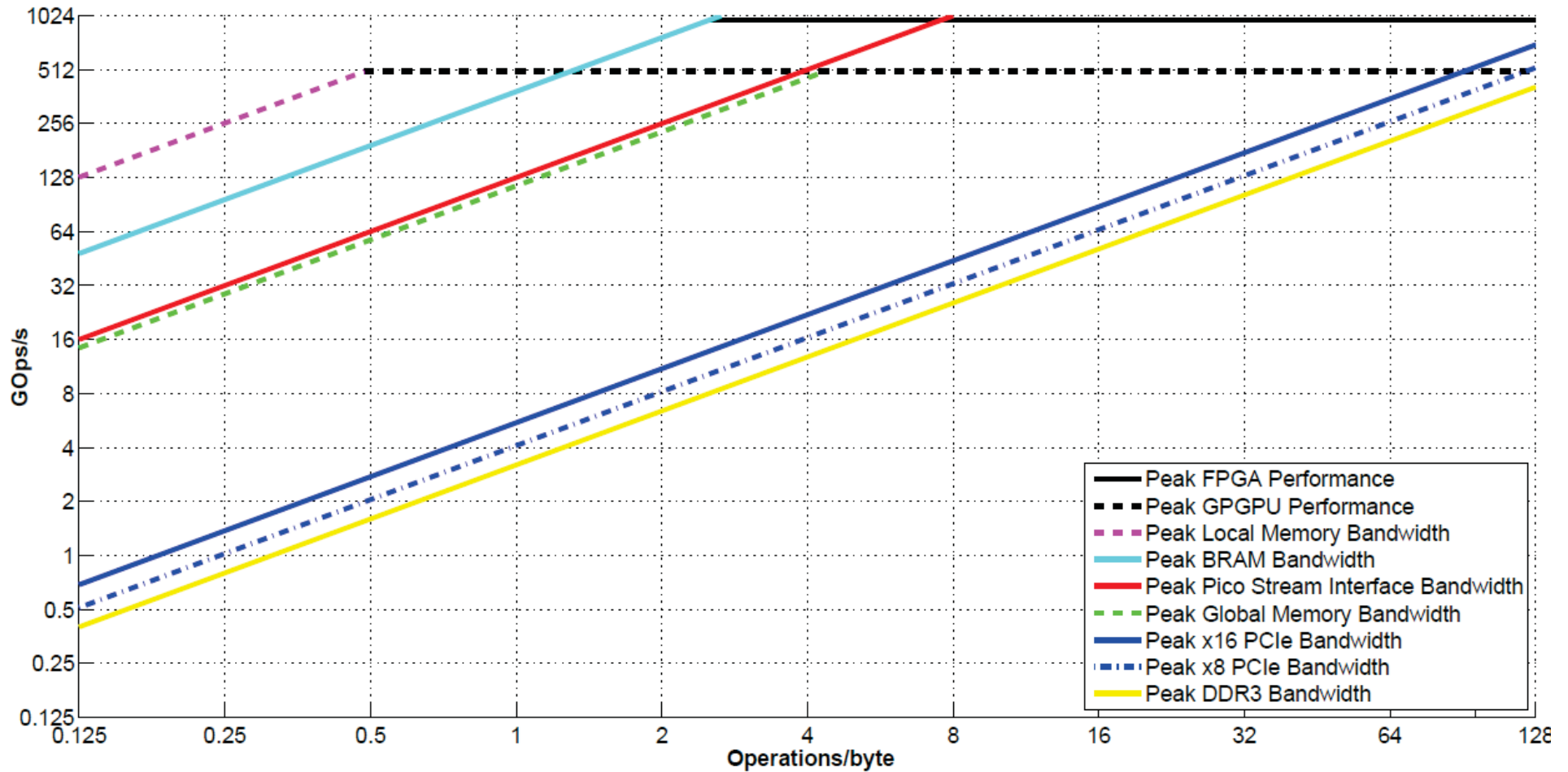
- Performance = Min(I/O dependent Perf., HW Peak Perf.)
- I/O dependent Perf. = **Ops/Byte** x **Bytes/s** = **CI** x **BW**  
where **CI** = Computational Intensity  
**BW** = I/O Bandwidth

# Roofline model:



- CI of algorithm  $\rightarrow$  results  $\rightarrow$  I/O or compute bound

# Superimposed GPU/FPGA roofline models for integer 32bits additions



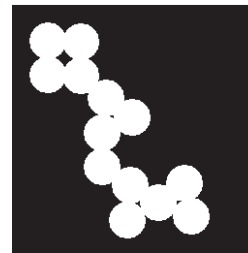
# 3. GPUs vs FPGAs

Comparing an image processing algorithm

# Implementing a morphological operation

- Basic morphological operation: Erosion 3x3

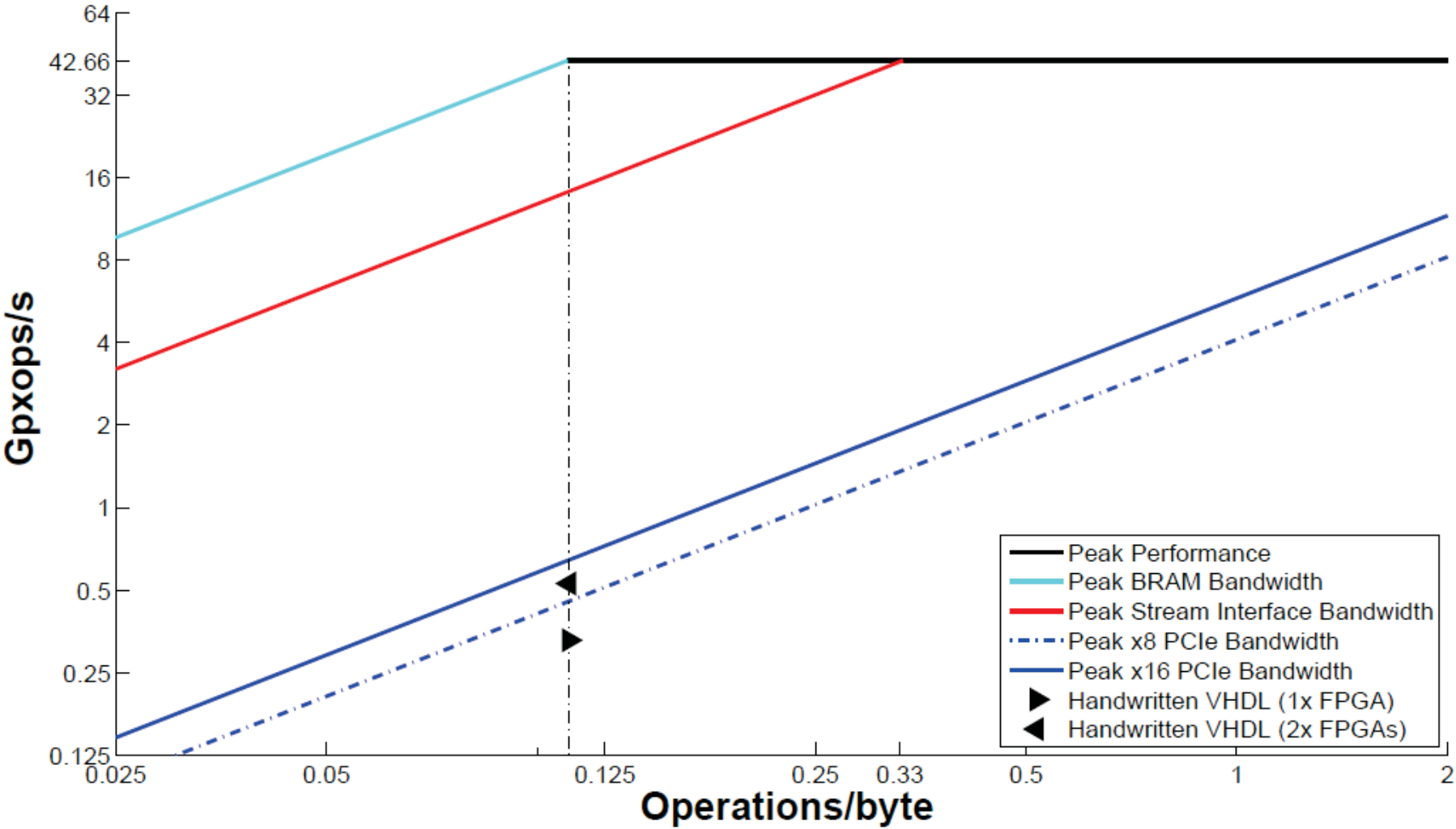
0 0 0 0 0		0 0 0 0 0
0 1 1 1 0		0 0 0 0 0
0 1 1 1 0	=>	0 0 1 0 0
0 1 1 1 0		0 0 0 0 0
0 0 0 0 0		0 0 0 0 0



- First implementation: handwritten code
- Second implementation: HLS-compilers



# Roofline model of erosion: Handwritten VHDL code



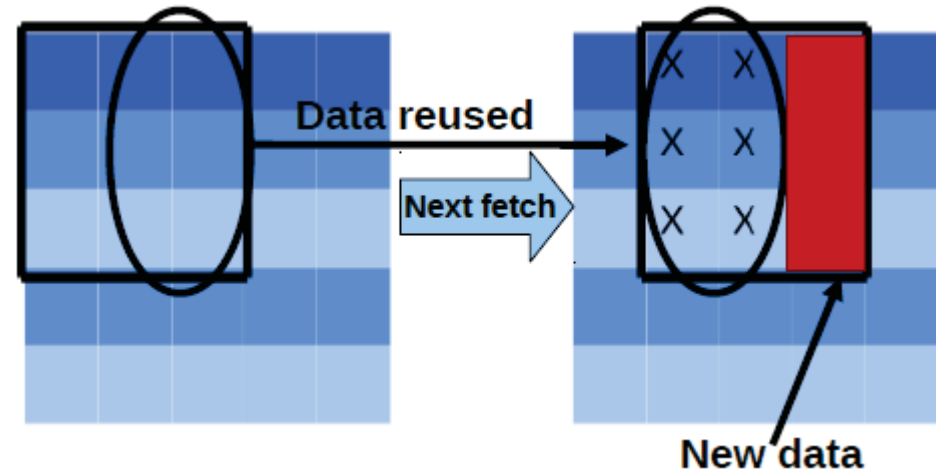
# Implementing a morphological operation with ROCCC (Riverside Optimizing Configurabe Compiler)

- Why ROCCC?:
  - Open Source
  - Stream oriented
  - Optimization to decrease memory accesses:
    - Smart Buffers
    - Partial Loop Unrolling

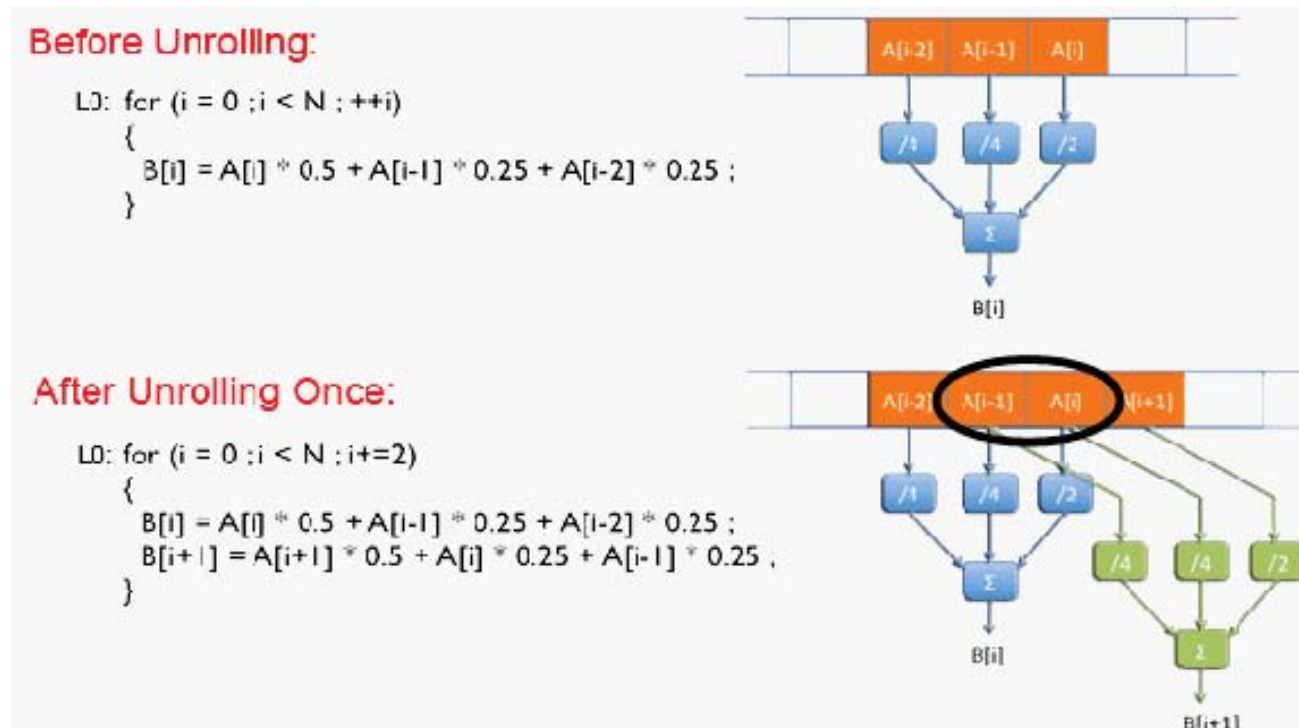
# Smart Buffers

- The compiler analyses the array access looking for possible reuse between loop iterations to reduce the number of off-chip memory accesses.

```
for(i = 0 ; i < 5 ; ++i )  
{  
  for (j = 0 ; j < 5; ++j)  
  {  
    row1 = A[i][j] + A[i][j+1] + A[i][j+2] ;  
    row2 = A[i+1][j] + A[i+1][j+1] + A[i+1][j+2] ;  
    row2 = A[i+2][j] + A[i+2][j+1] + A[i+2][j+2] ;  
    B[i][j] = row1 + row2 + row3 ;  
  }  
}
```

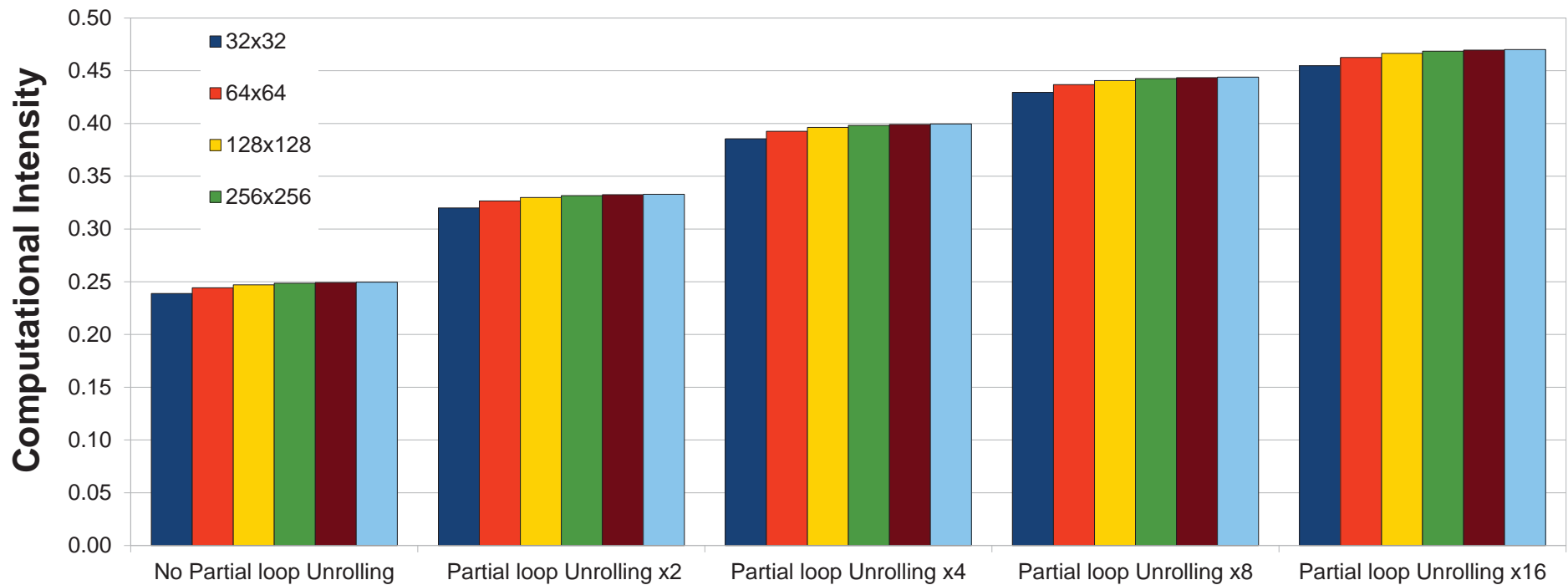


# Partial loop unrolling

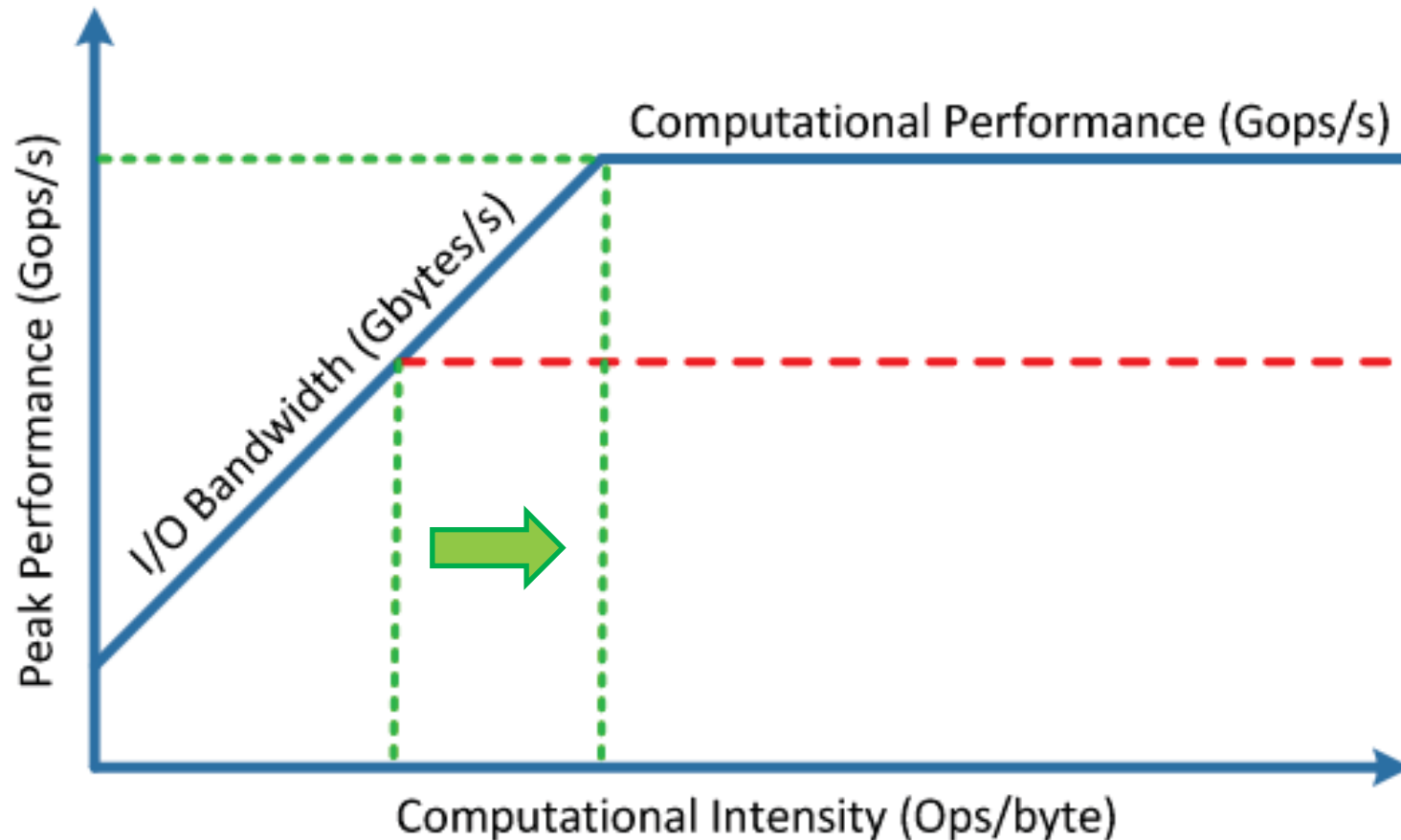


- In ROCCC, an output stream channel must be defined and the outputs must be multiplexed in time.

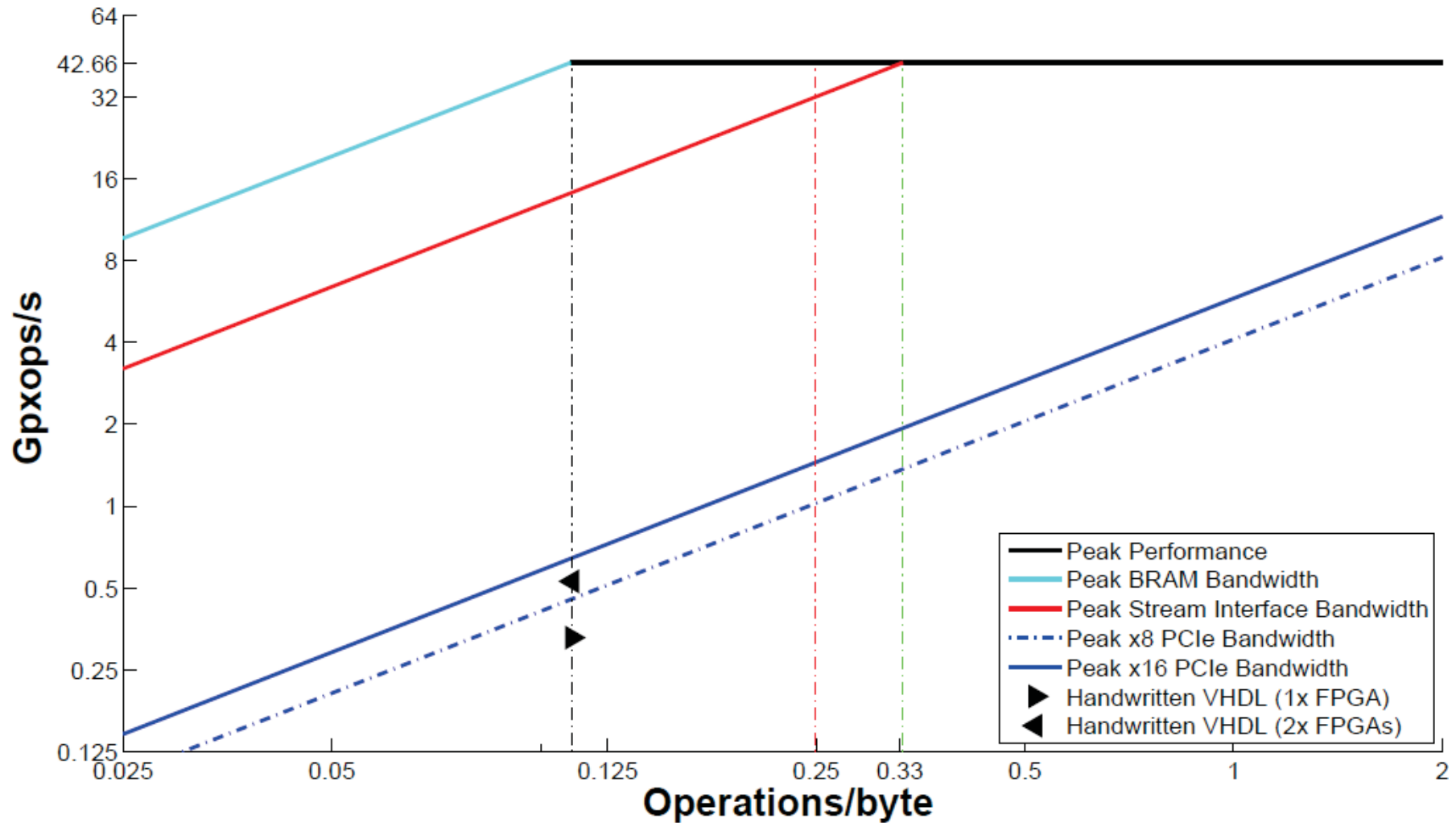
# Impact of the smart buffers and the partial loop unrolling over the Computational Intensity



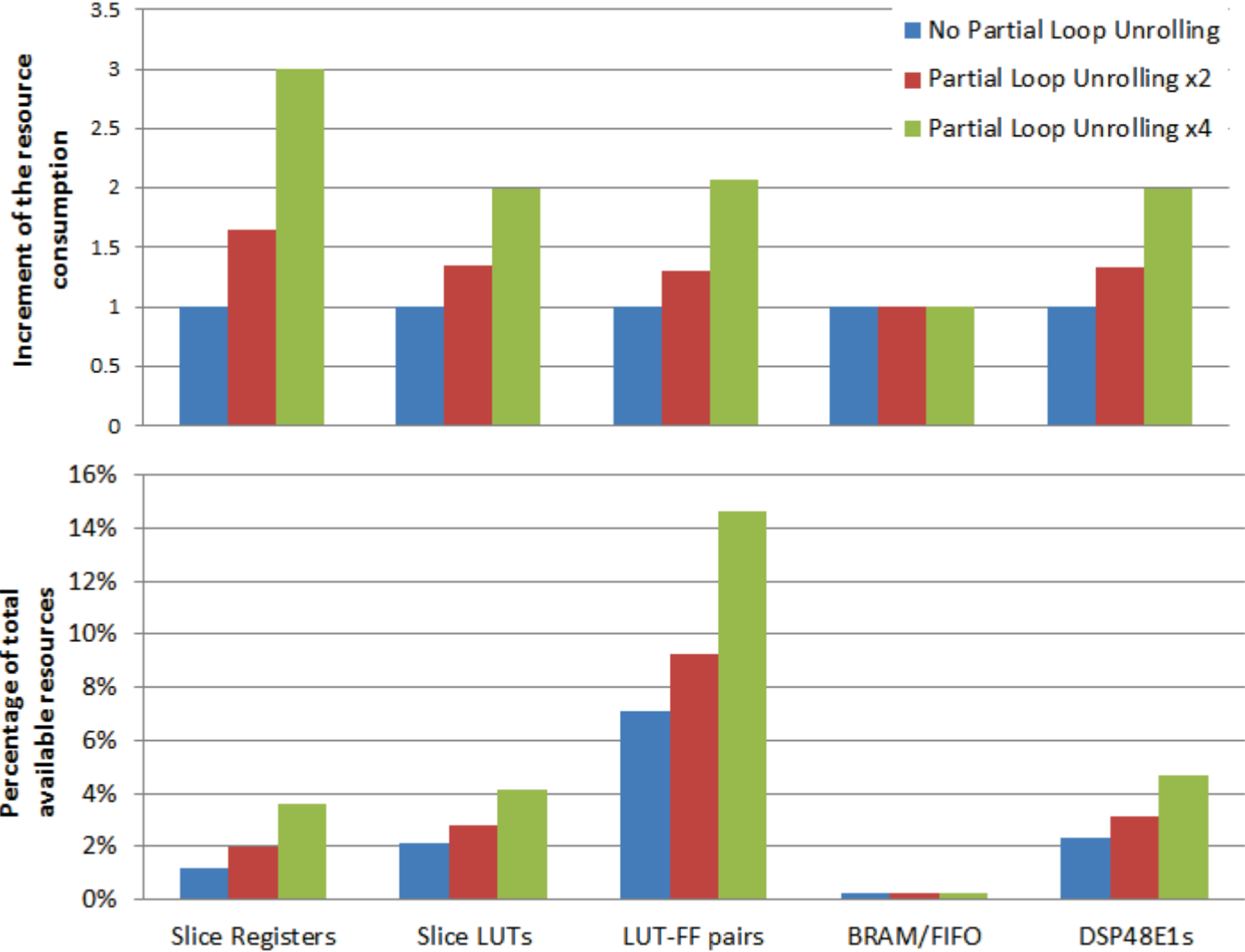
# Improving performance by increasing the Computational Intensity



# Roofline model of erosion: Increasing the original Computational Intensity

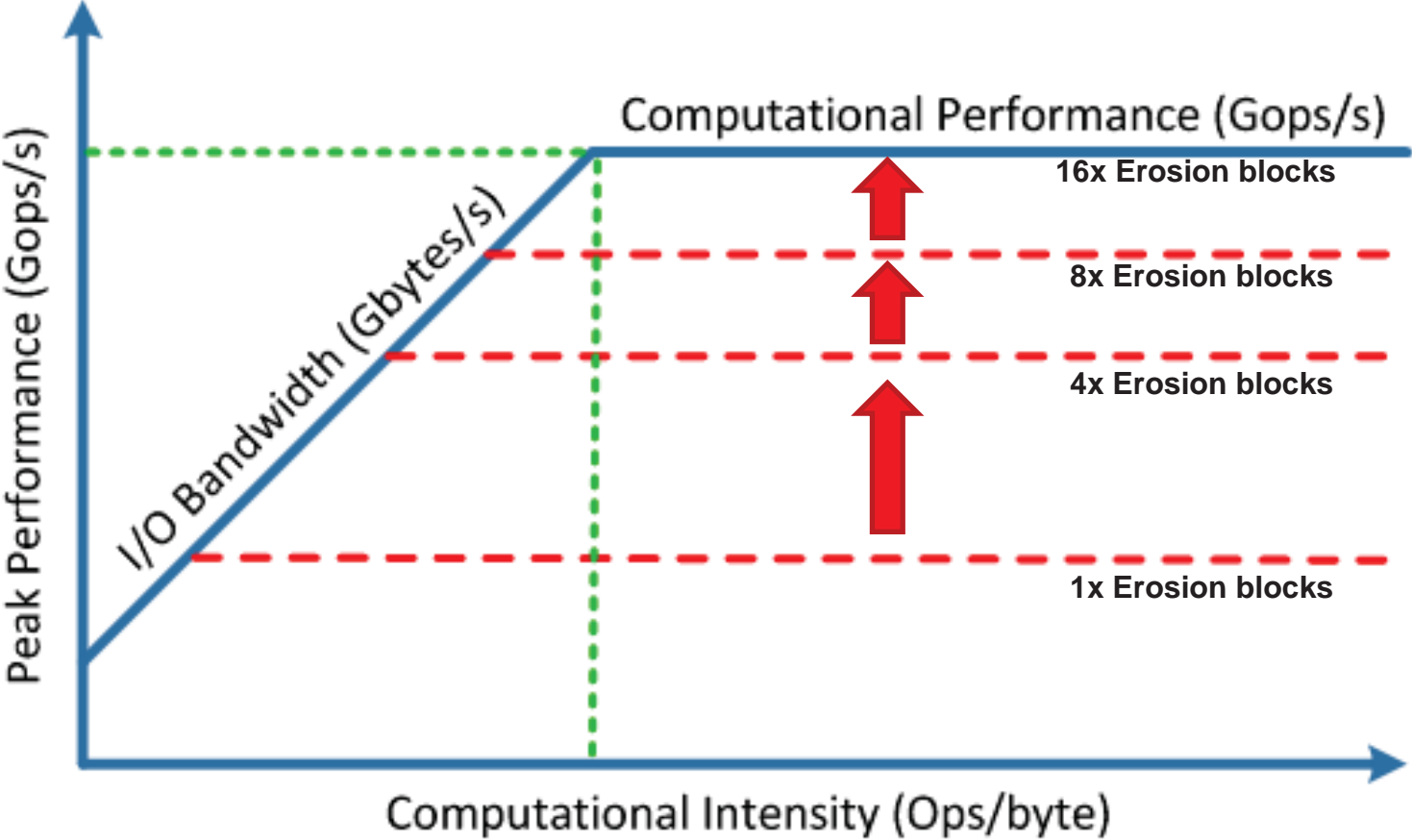


# Resource Consumption

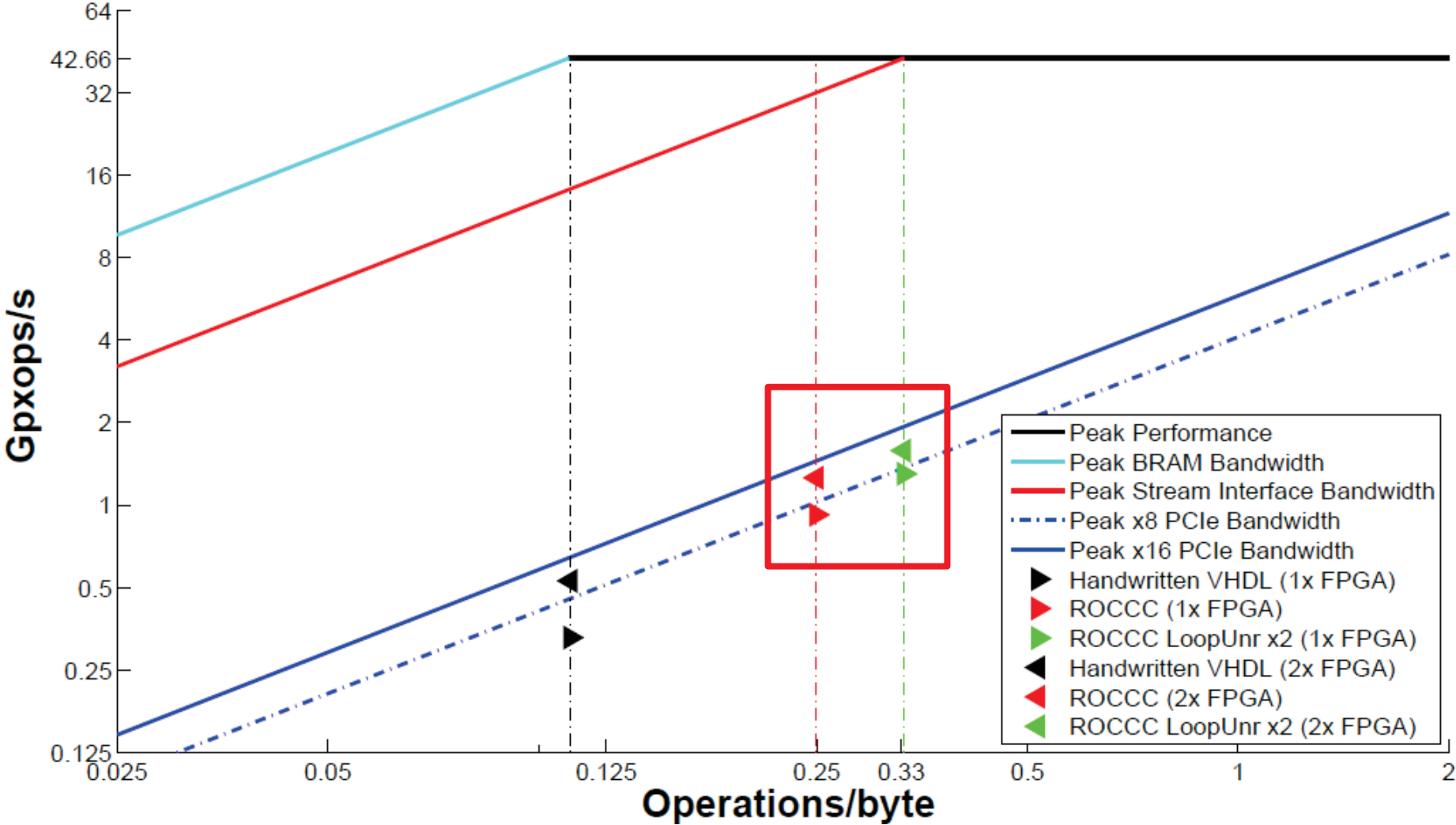




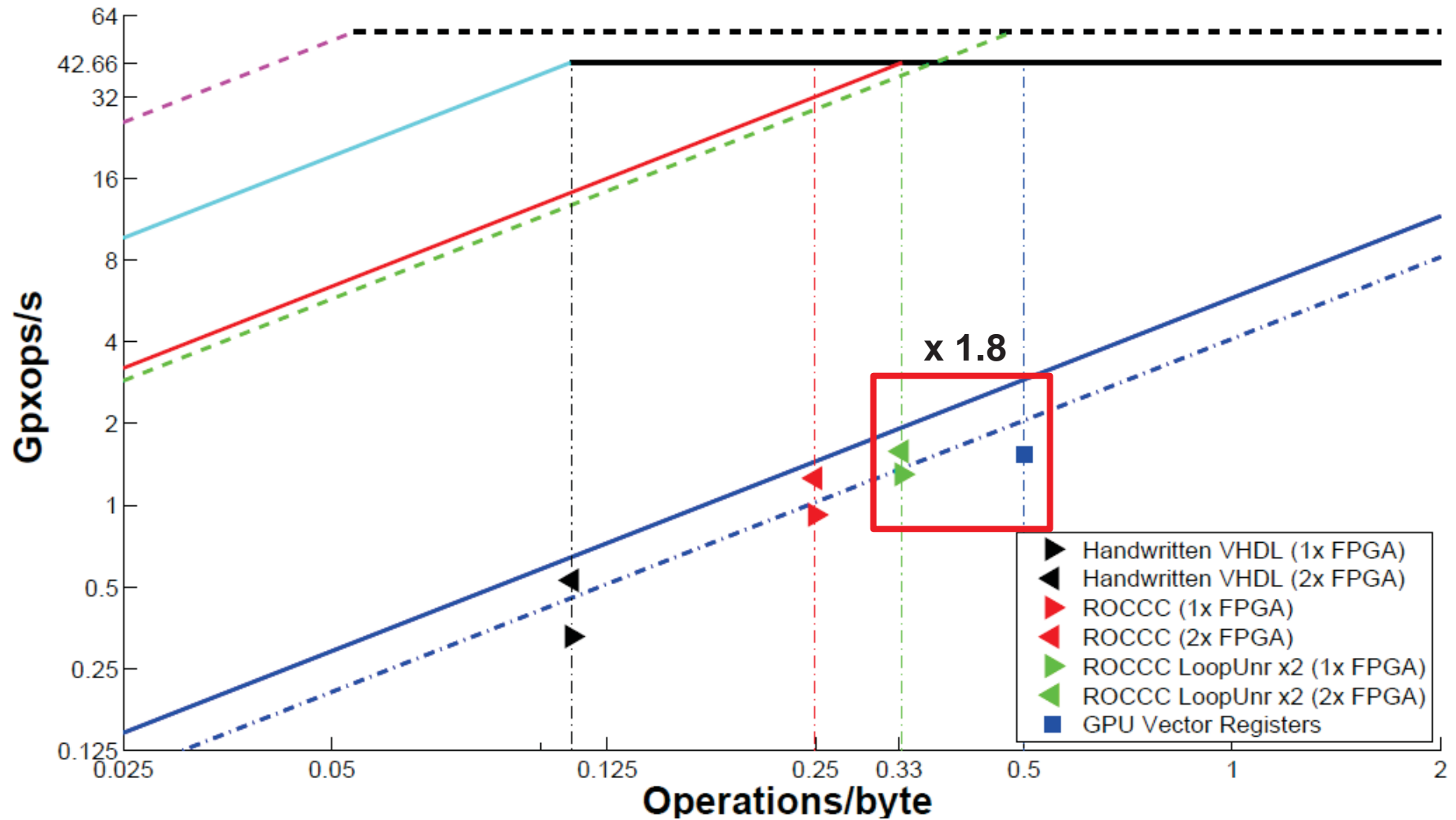
# Improving performance by increasing parallelism



# Roofline Model: Handwritten VHDL code vs ROCCC



# GPU vs FPGA Performance

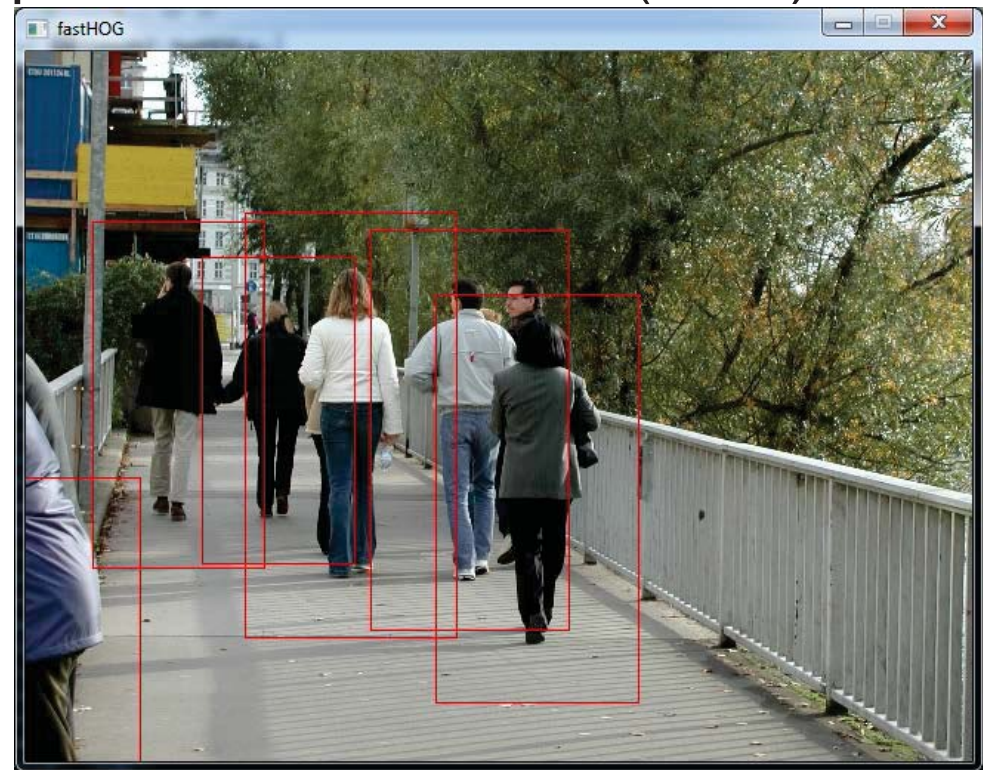


# 4. Combining GPUs and FPGAs

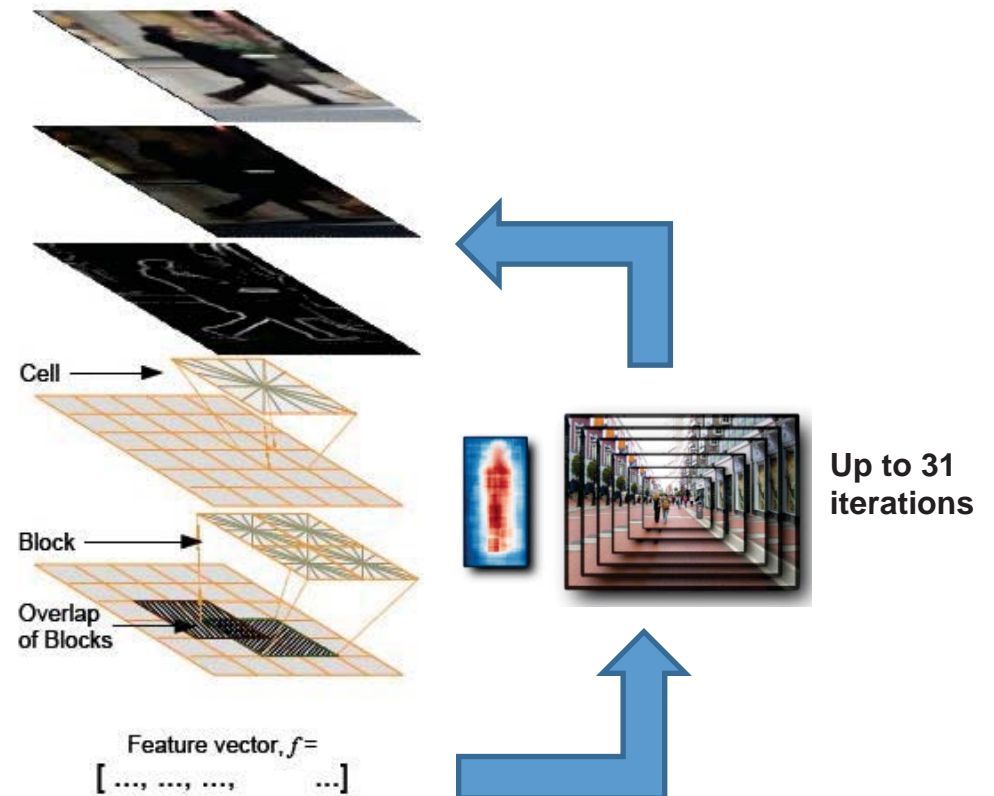
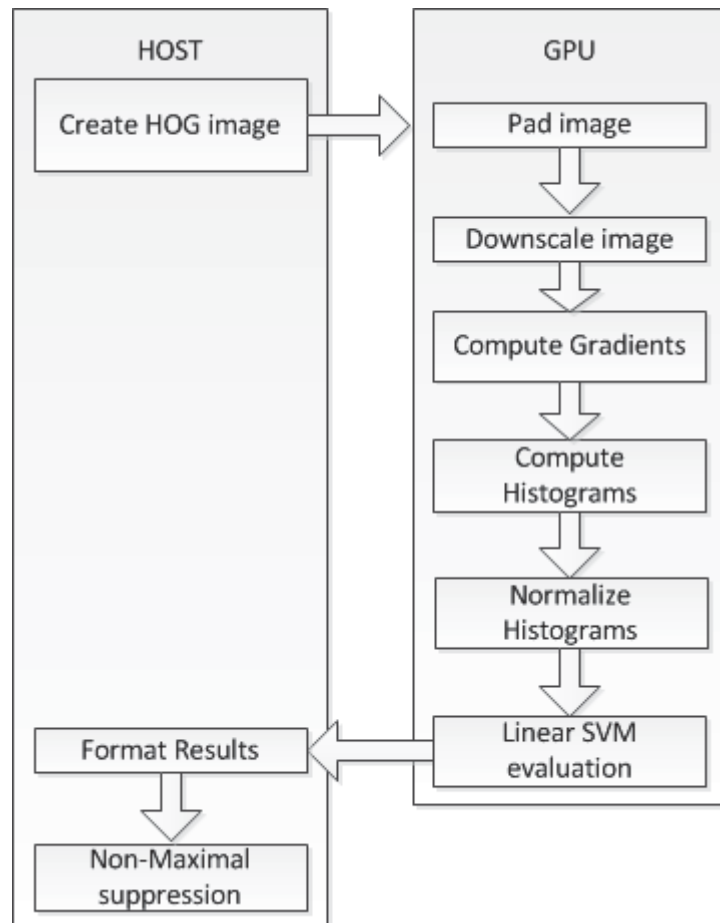
Exploiting the best of both technologies

# Pedestrian detection: fastHOG

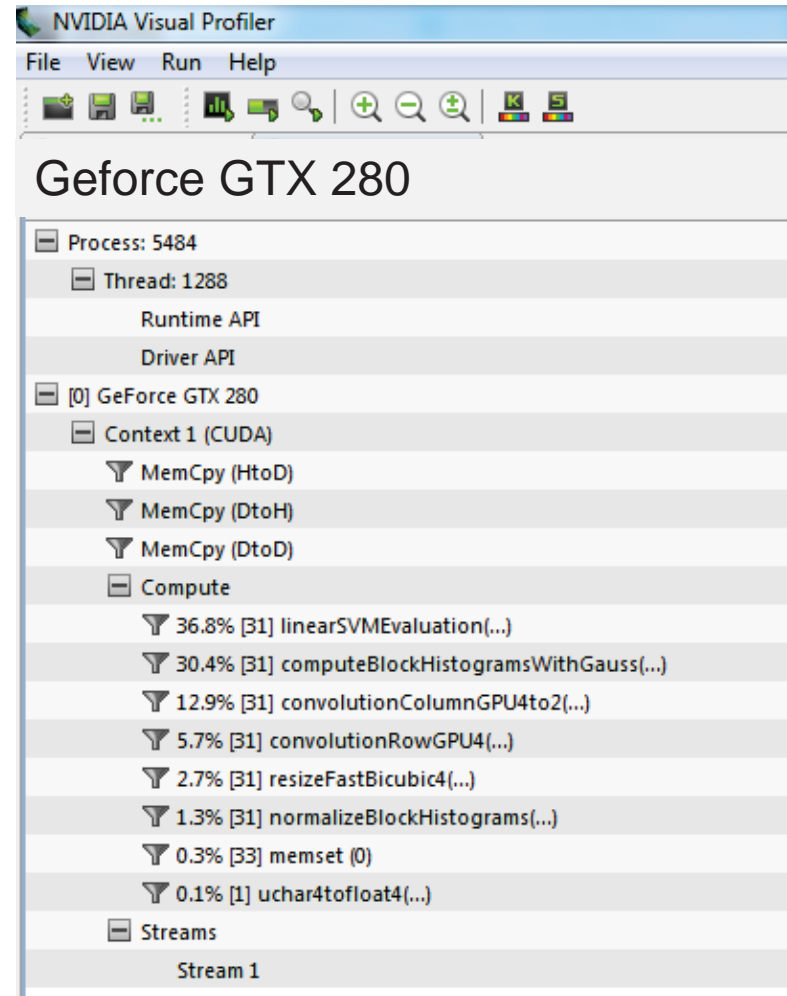
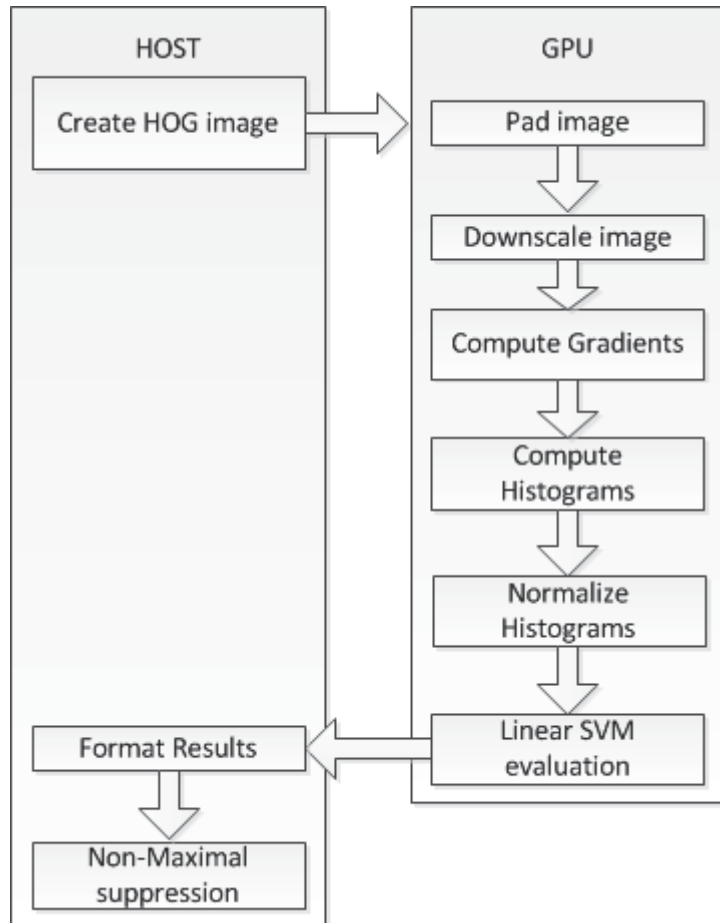
- Detecting people in images using Histograms Oriented Gradients (HOG) and Support Vector Machines (SVM).
- Different Steps
  - Some ideal for GPU
  - Others ideal for FPGA
- Existing GPU version called fastHOG.



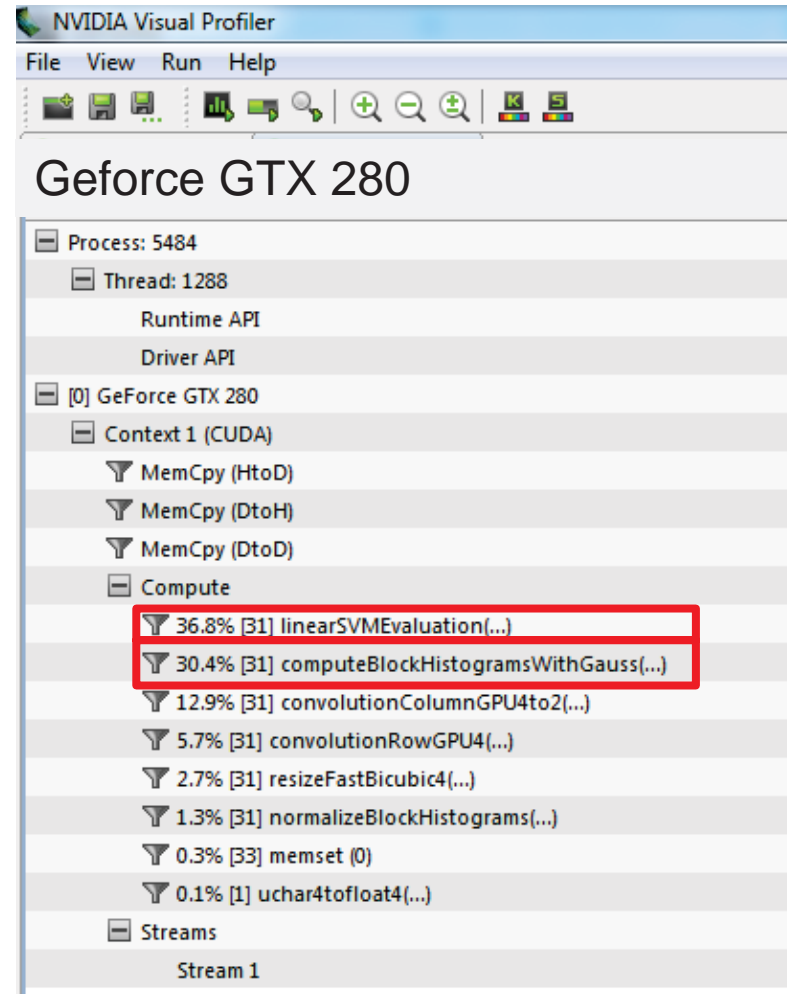
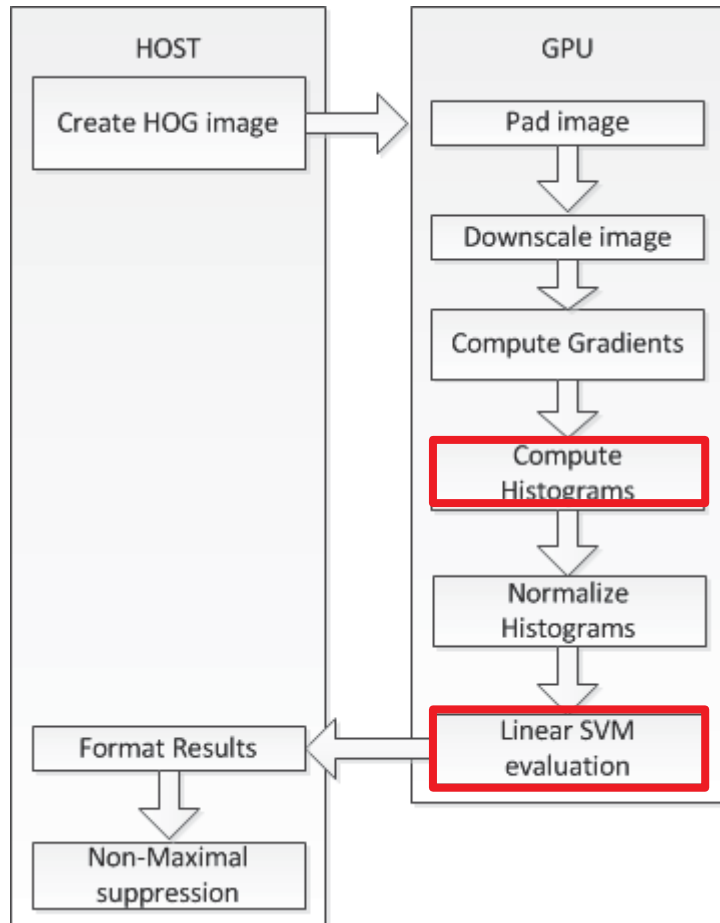
# fastHOG: HOG + SVM



# Identifying the candidate to accelerate: Histogram + SVM Computation

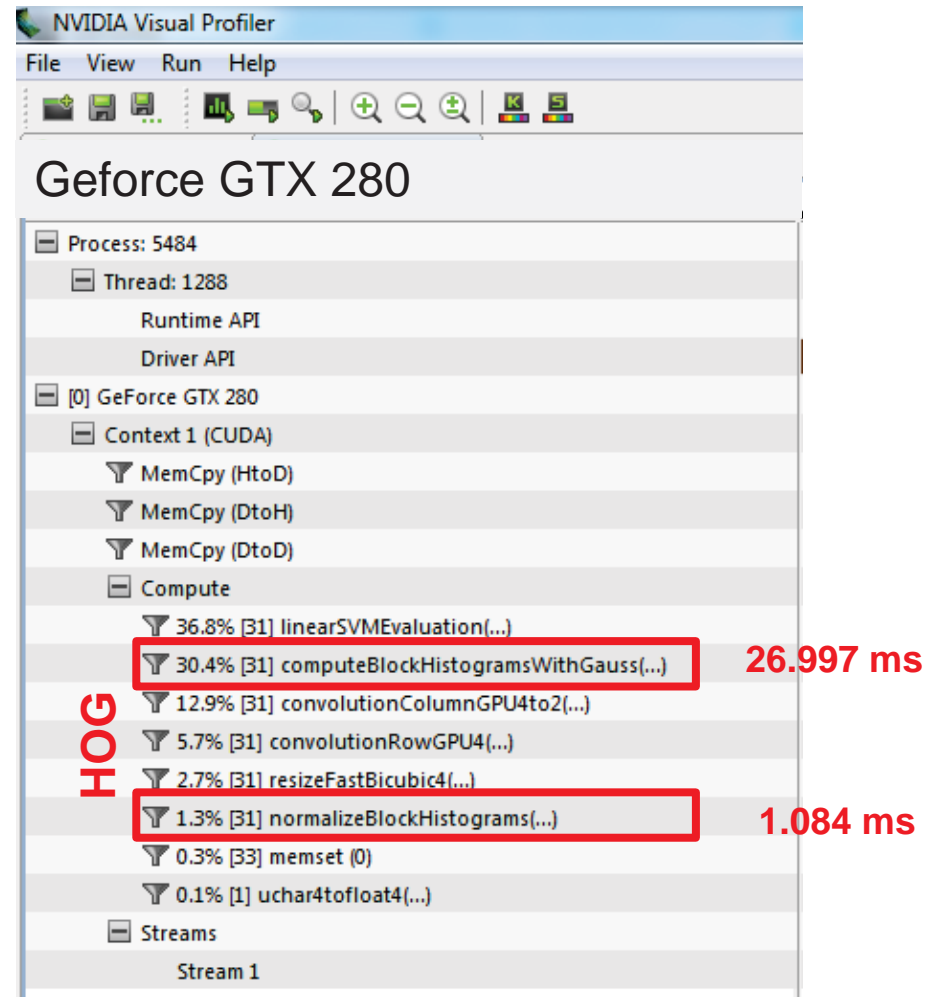
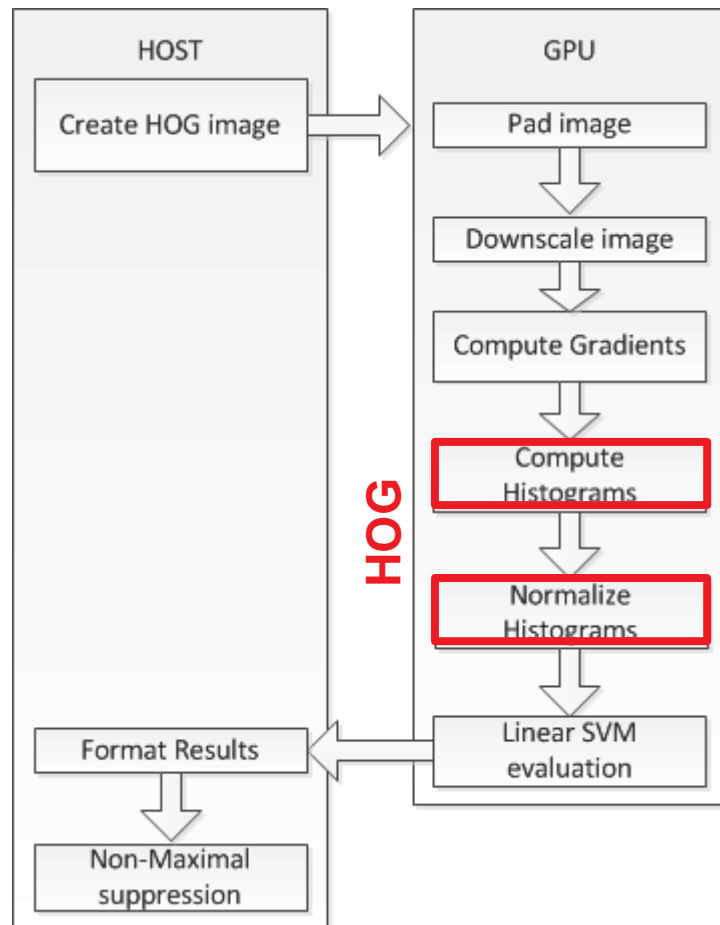


# Identifying the candidate to accelerate: Histogram + SVM Computation

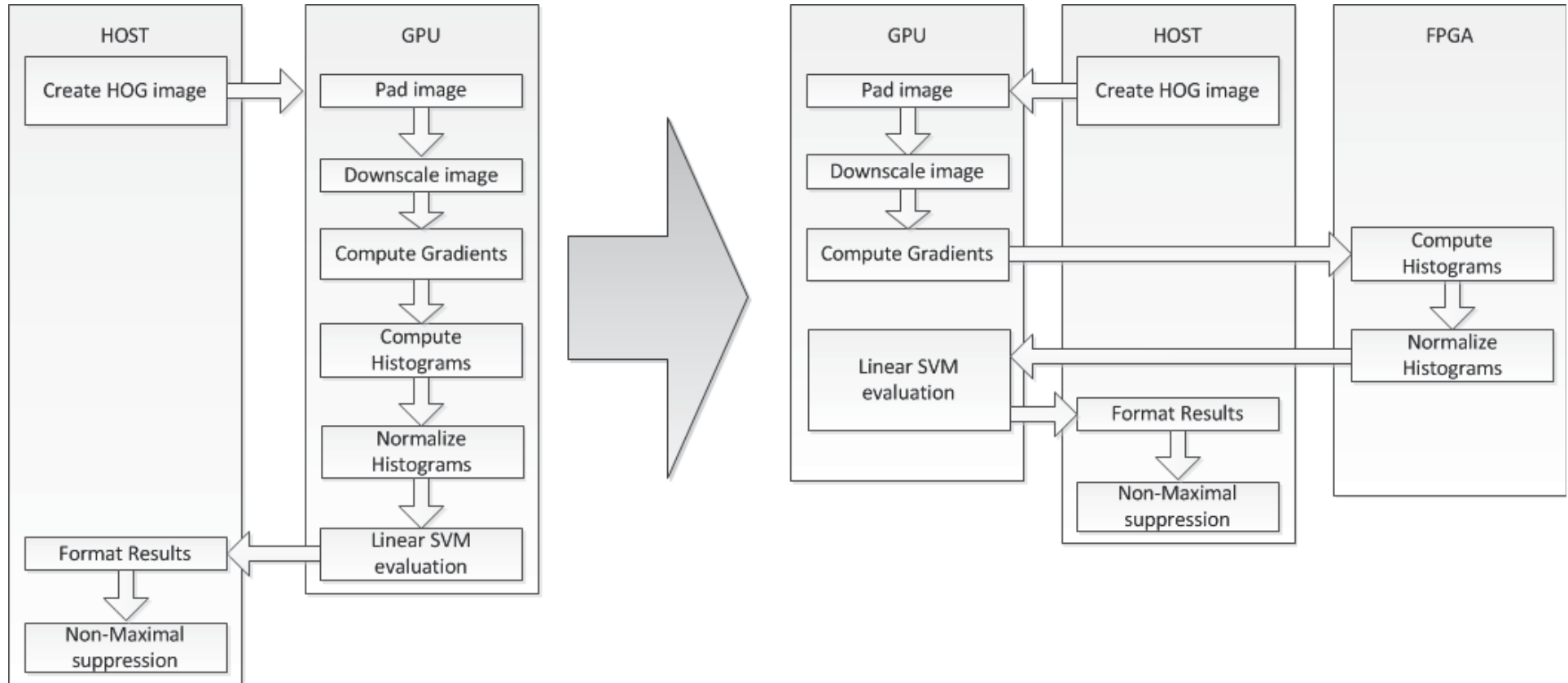




# Identifying the candidate to accelerate: Histogram Computation and Normalization



# New dataflow computing HOG on the FPGA



# Adapting the code for FPGAs

- Mathematical types, functions and footprint:
  - Floating point → Fixed point
  - Adapt mod, floor, divisions and other computationally expensive operations
  - Adjust the data bit-width: performance vs accuracy
- Memory use and reuse:
  - Rewrite the code to reduce memory accesses
- VivadoHLS Directives:
  - Pipelining, Stream interface, Partial Loop Unrolling
  - Impact of the clock definition over the design.

# Adapting the code working @ 125MHz

## Floating point

- Latency:  
160,128,522 clock cycles

- Resource consumption:

	BRAM 18K	DSP48	FF	LUT	SLICE
<b>Total</b>	17	50	6048	6930	0
<b>Available</b>	832	768	301440	150720	37680
<b>Utilization (%)</b>	2	6	2	4	0

## Fixed point

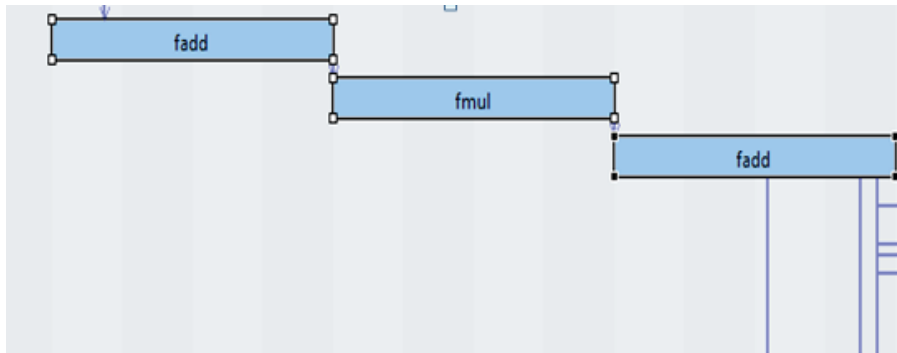
- Latency:  
61,055,286 clock cycles

- Resource consumption:

	BRAM 18K	DSP48	FF	LUT	SLICE
<b>Total</b>	16	62	6071	11821	0
<b>Available</b>	832	768	301440	150720	37680
<b>Utilization (%)</b>	1	8	2	7	0

# Adapting the code

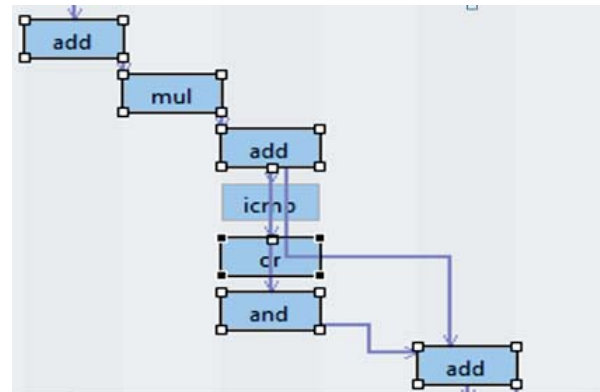
## Floating point



- Operation Latency:  
12 clock cycles

- It is important to avoid divisions, mod or floor operations due to its high latency and resource consumption.

## Fixed point



- Operation Latency:  
5 clock cycles

# Adapting the data bit-width

- Knowledge of the input data range.
- The input gradients are composed of two parameters:
  - Magnitude: between  $[0, 17]$   $\longrightarrow$  5 bits for the integer part
  - Orientation: between  $[-179, 180]$   $\longrightarrow$  9 bits
- The resource consumption as well as the accuracy is decreased due to the fixed point conversion.
- However, it allows to place more blocks in parallel and to exploit the I/O bandwidth.

# Adapted HOG code: Main function

```
void FullStreamHOG
(
    fixed_t2 gradients[MaxGradients],
    unsigned char noBlocksX,
    unsigned char noBlocksY,
    fixed_t histograms_norm[MaxHistograms]
){
    // Initialize internal memory
    fixed_t histograms0[36], histograms1[36], histograms2[36], histograms3[36];
    fixed_t_Norm histograms[36];

    // Initialization
    for (int i = 0; i < 36; ++i) {
        histograms0[i] = 0;
        histograms1[i] = 0;
        histograms2[i] = 0;
        histograms3[i] = 0;
    }

    unsigned int gradientStart = 0;
    unsigned int histogramStart = 0;


    LY:for (int blockY = 0; blockY < noBlocksY; ++blockY) {
        LX:for (int blockX = 0; blockX < noBlocksX; ++blockX) {

            // Histogram computation
            StreamHOG(&gradients[gradientStart], histograms0, histograms1, histograms2, histograms3);

            // Histogram normalization
            normalizeHistograms(histograms0, histograms1, histograms2, histograms3, &histograms_norm[histogramStart]);

            gradientStart += 16*16;
            histogramStart += blockSizeX*blockSizeY*noHistogramBins;
        }
    }
}
```

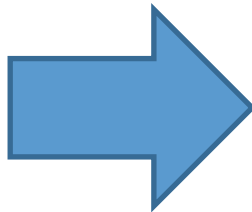
Memory  
Rearrangement



# Impact of the directives: Pipelining the code

```

● StreamHOG
  x+/ L0
  x+/ L1
  % HLS PIPELINE
● normalizeHistograms
● FullStreamHOG
  ● gradients
  % HLS INTERFACE ap_fifo port=gradients
  ● noBlocksX
  ● noBlocksY
  ● histograms_norm
  % HLS INTERFACE ap_fifo port=histograms_norm
  x[] histograms0
  x[] histograms1
  x[] histograms2
  x[] histograms3
  x[] histograms
  x+/ for Statement
  x+/ LY
  % HLS LOOP_TRIPCOUNT tripcount min=0 max=92
  x+/ LX
  % HLS LOOP_TRIPCOUNT tripcount min=0 max=69
  % HLS PIPELINE
  
```



- High Latency:  
43,801,238 clock cycles

- High Resource consumption

	BRAM 18K	DSP48	FF	LUT	SLICE
<b>Total</b>	132	454	62654	146792	0
<b>Available</b>	832	768	301440	150720	37680
<b>Utilization (%)</b>	<b>15</b>	<b>59</b>	<b>20</b>	<b>97</b>	<b>0</b>

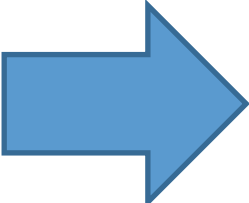
- Pipeline the full code is not always the best option.



# Impact of the directives: Partial pipelining

```

● StreamHOG
  L0
  L1
  % HLS PIPELINE
● normalizerHistograms
  x[1] HistoTemp
  x[1] HistoTemp2
  L[1] L[1]1
  % HLS PIPELINE
  Mul0
  Sqrt1
  L[1] L[1]2
  % HLS PIPELINE
  Mul11
  Mul12
  Sqrt2
  L[1] L[1]3
  % HLS PIPELINE
  Mul2
● FullStreamHOG
  ● gradients
  % HLS INTERFACE ap_fifo port=gradients
  ● noBlocksX
  ● noBlocksY
  ● histograms_norm
  % HLS INTERFACE ap_fifo port=histograms_norm
  x[1] histograms0
  x[1] histograms1
  x[1] histograms2
  x[1] histograms3
  x[1] histograms
  for Statement
  LY
  % HLS LOOP_TRIPCOUNT tripcount min=0 max=92
  LX
  % HLS LOOP_TRIPCOUNT tripcount min=0 max=69
  
```



- Latency:  
7,954,266 clock cycles

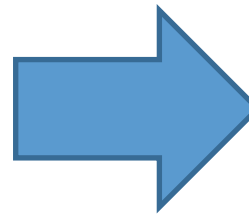
- Resource consumption:

	BRAM 18K	DSP48	FF	LUT	SLICE
<b>Total</b>	16	65	6078	11664	0
<b>Available</b>	832	768	301440	150720	37680
<b>Utilization (%)</b>	<b>1</b>	<b>8</b>	<b>2</b>	<b>7</b>	<b>0</b>

# Impact of the directives: Partial Loop Unrolling

```

● FullStreamHOG
  ● gradients
  %o HLS INTERFACE ap_fifo port=gradients
  ● noBlocksX
  ● noBlocksY
  ● histograms_norm
  %o HLS INTERFACE ap_fifo port=histograms_norm
  x[] histograms0
  x[] histograms1
  x[] histograms2
  x[] histograms3
  x[] histograms
  *+//
  :? LY
  %o HLS LOOP_TRIPCOUNT tripcount min=0 max=92
  *+//
  :? LX
  %o HLS LOOP_TRIPCOUNT tripcount min=0 max=69
  %o HLS UNROLL factor=2
  
```



- Latency:  
8,069,910 clock cycles

- Resource consumption:

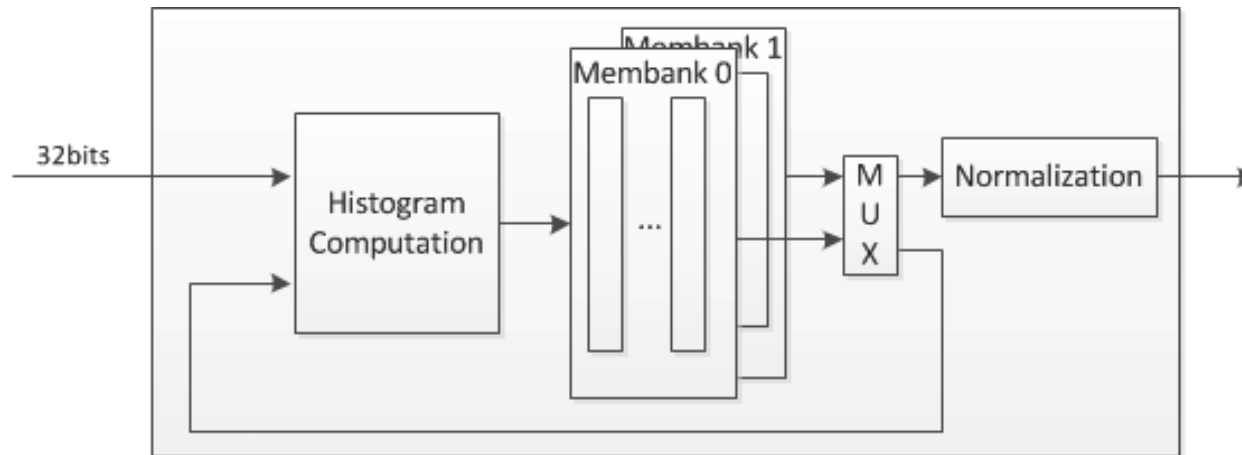
	BRAM 18K	DSP48	FF	LUT	SLICE
<b>Total</b>	16	65	6137	11735	0
<b>Available</b>	832	768	301440	150720	37680
<b>Utilization (%)</b>	1	8	2	7	0

# Impact of the clock definition on VivadoHLS

- Once the clock of the system has been defined (mandatory to synthesize) the compiler would focus all the effort to achieve the expected frequency.
- That means a high resource consumption and a extremely low latency.
- Specially when several directives as pipelining are applied.
- The best strategy is to define the operational frequency from the beginning.

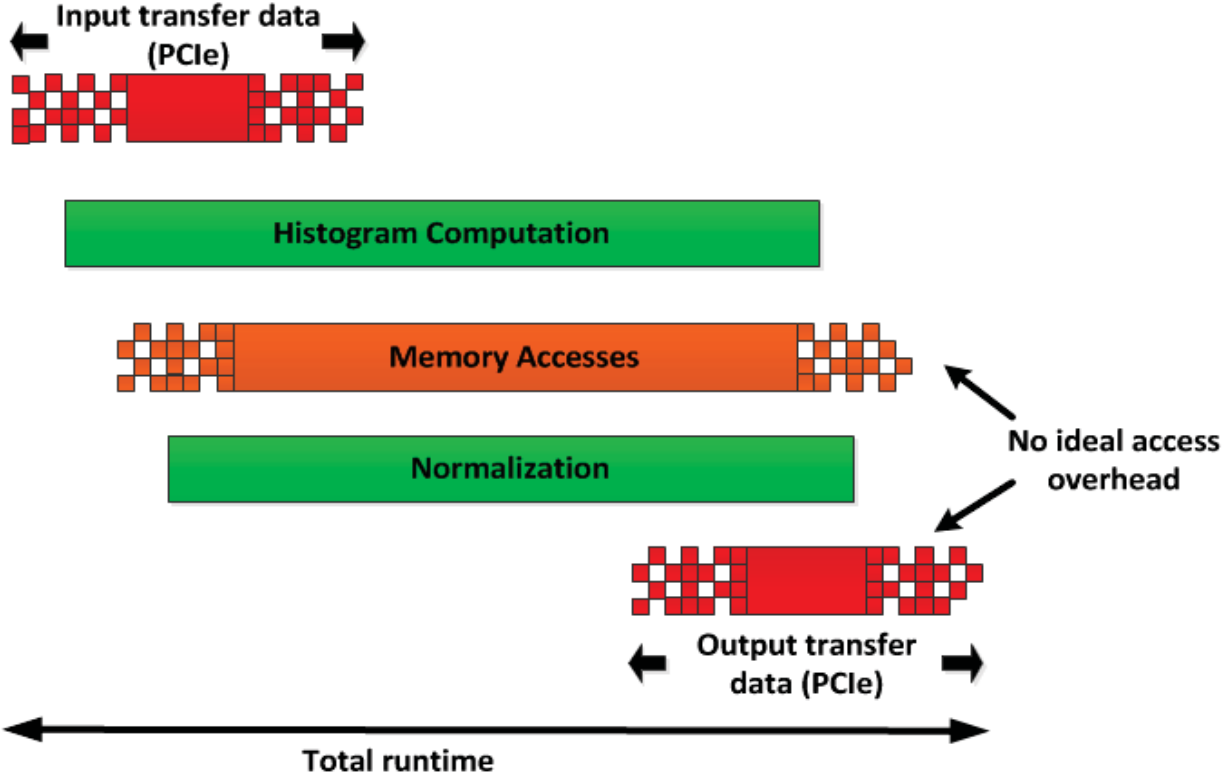
# HOG implementation

- Pipeline solution including the normalization part

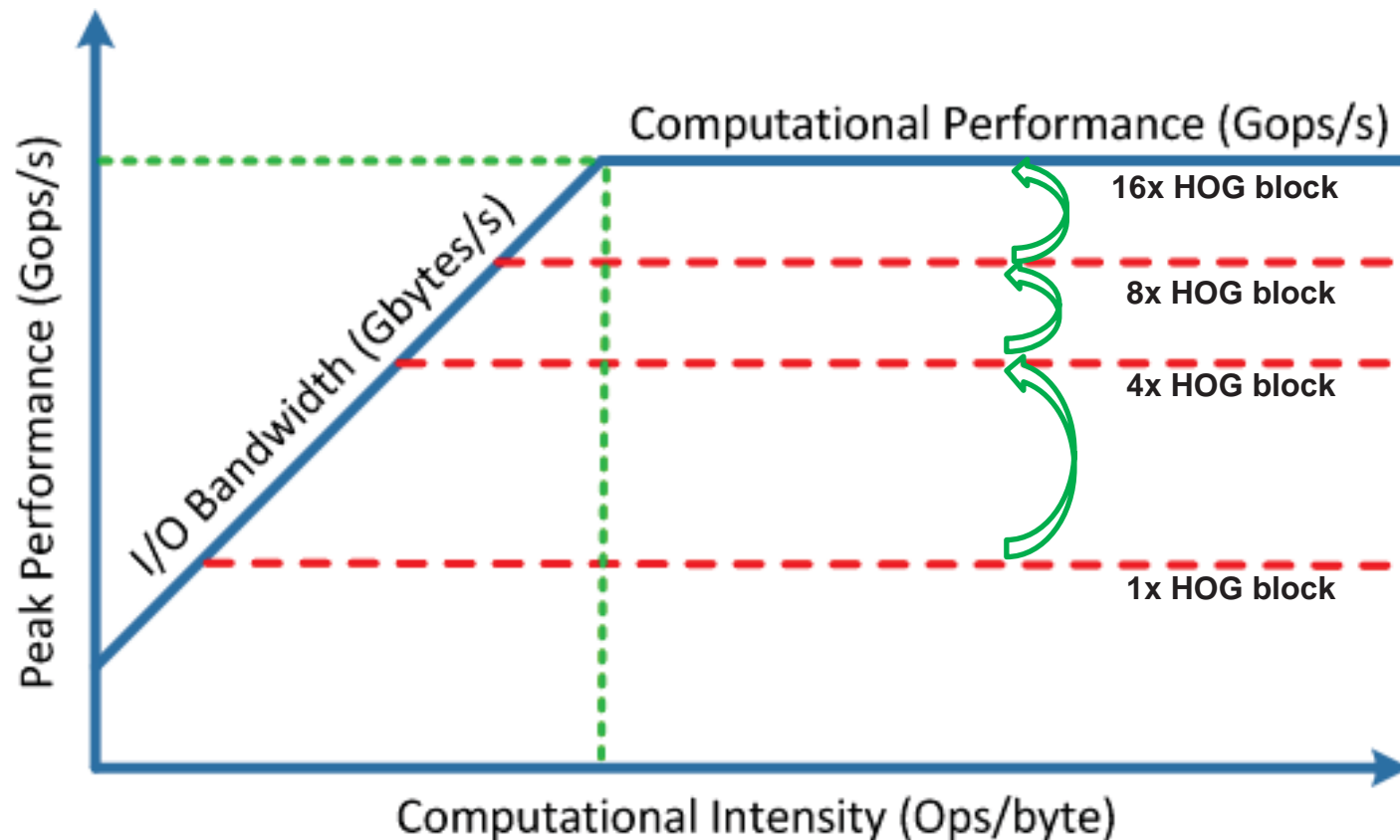


- Latency drastically reduced (about x35):  
Original design: 2,1s → Final design: 61ms

# Execution on the FPGA: Streaming + Pipelining



# Improving performance by increasing the parallelism up to the maximum resources available on the FPGA



- One HOG block consumes less than 7% of logic resources

# Comparing FPGA/GPUs HOG computation

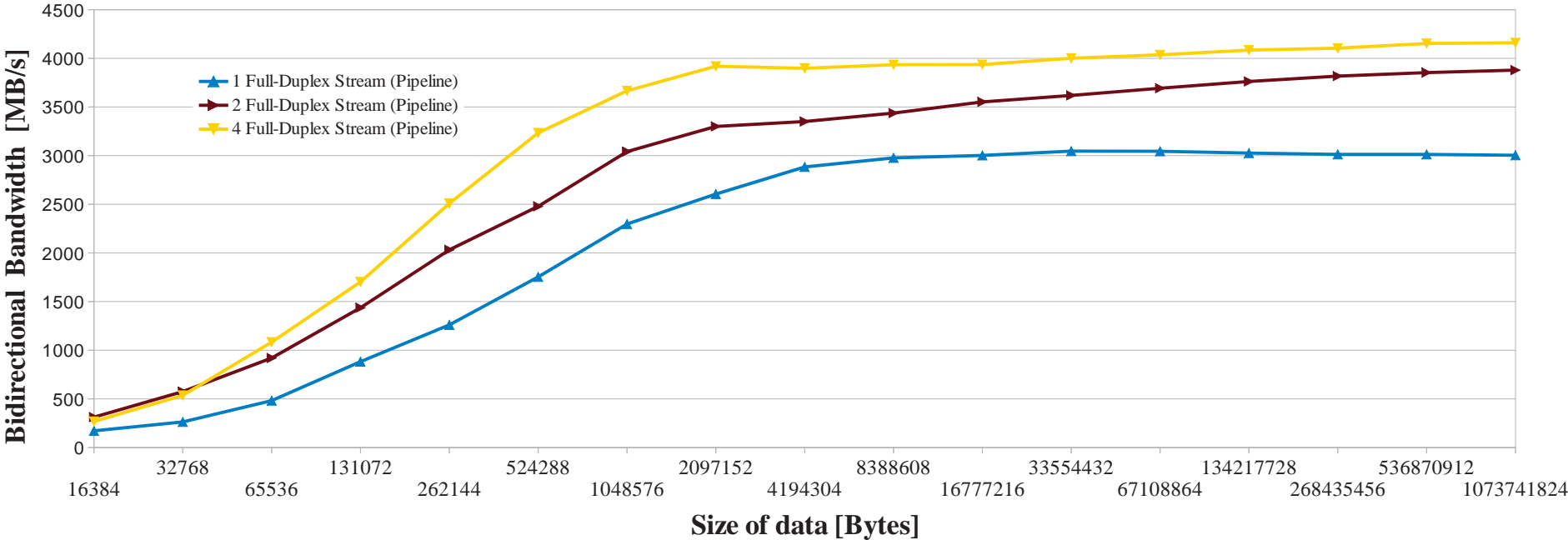
Iterations	Speed Up of the FPGA(s) over the Tesla C2050 GPU		Speed Up of the FPGA(s) over the Geforce GTX280 GPU	
	1xFPGA	2xFPGA	1xFPGA	2xFPGA
92x69	37%	68%	85%	93%
...	...	...	...	...
20x15	17%	59%	77%	88%
Average	x1.61	x3.22	x6.45	x13.69

# Detailed GPU/FPGA combination including their communication

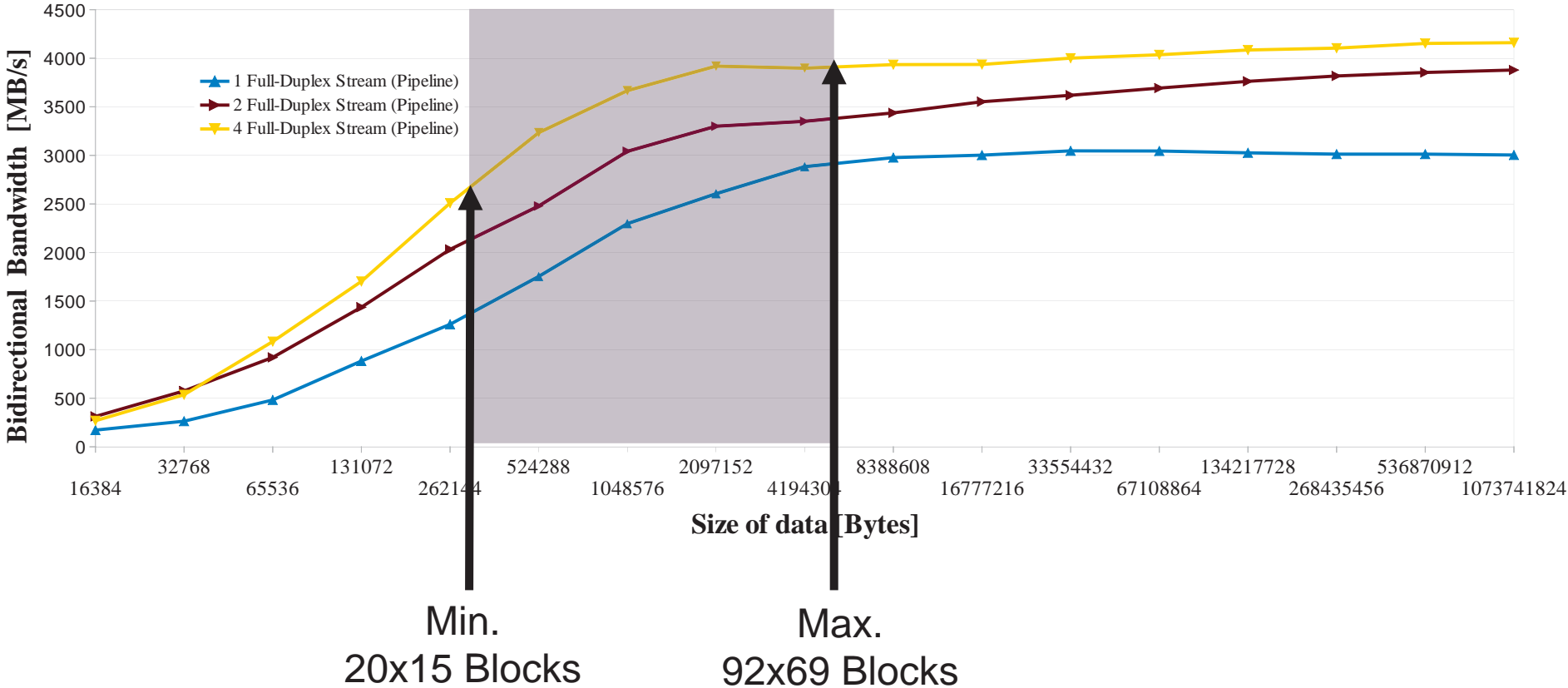




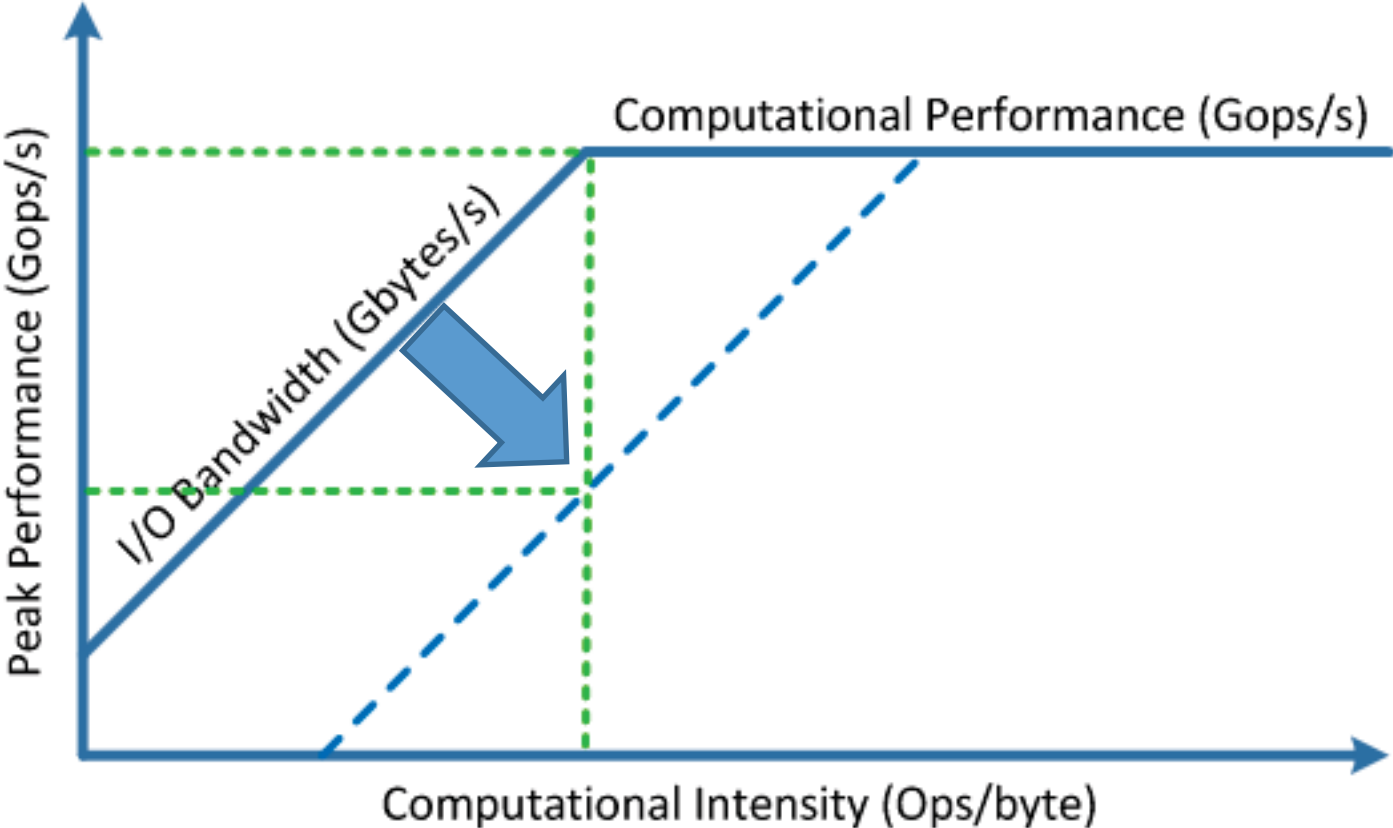
# Impact of the PCIe protocol overhead



# Impact of the PCIe protocol overhead



# Impact of the PCIe protocol overhead



# Performance combining GPU/FPGA

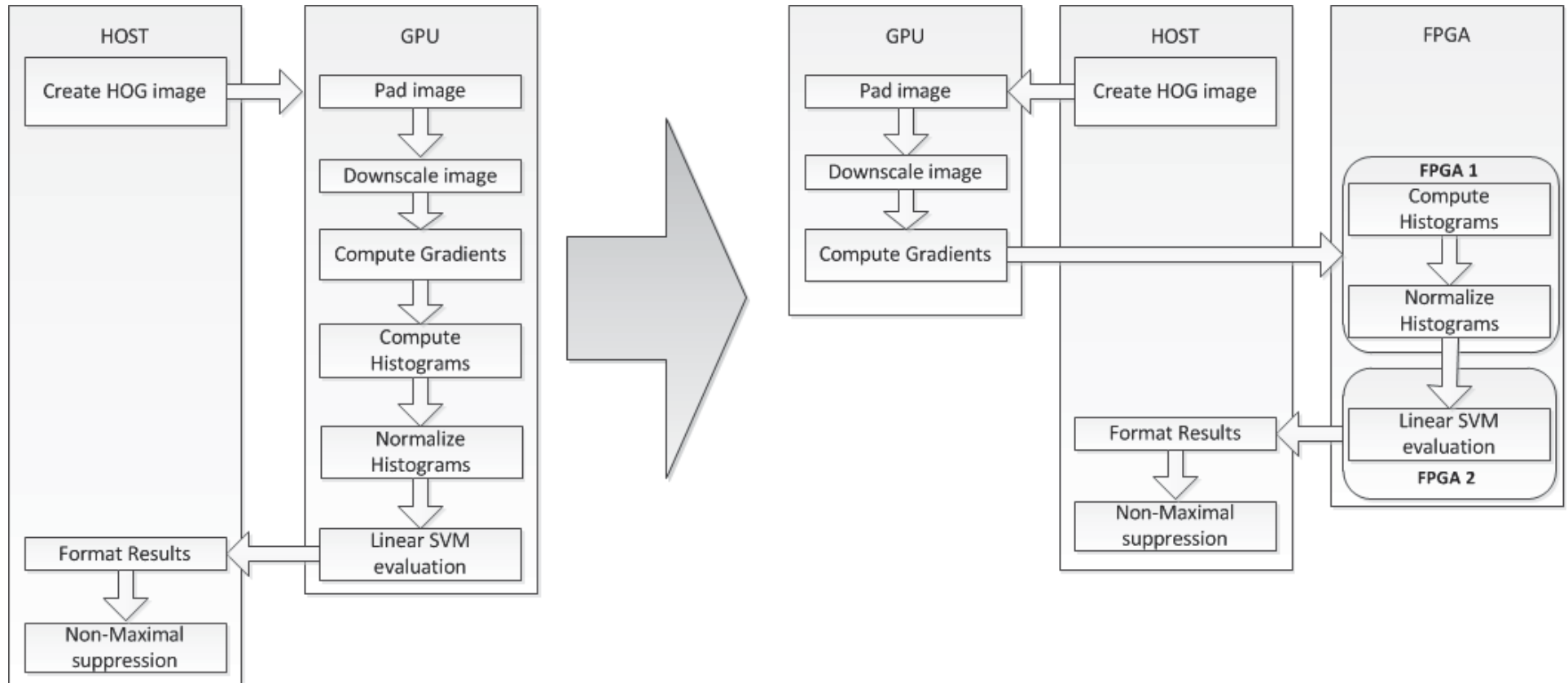
	<b>GPU*</b>	<b>GPU* + FPGA 16HOGs</b>	<b>GPU* + 2xFPGA 16HOGs</b>
<b>92x69</b>	6547	7260	5198
...	...	...	...
<b>20x15</b>	467	1846	1653
<b>Total Execution [ms]</b>	73066	108738	86208

\* Tesla C2050

# So, when to combine?

- In our case, when the FPGA implementation speed up the design more than 60% compared to the GPU.
- And when the amount of data to transfer is higher enough to reach the maximum PCIe bandwidth.

# Exploiting our modular Pico Board: HOG + SVM



# 5. Conclusions

# Conclusions of our HLS experience

- For algorithms with low CI, partial loop unrolling and other optimizations (smart buffers) are able to increase the CI have obtained higher performance.
- For algorithms with high CI, the most important is the resource consumption, which determinates the maximum realizable parallelism.
- In both cases, to exploit the FPGA's features it is recommended to pipeline the stages and to stream the I/O.
- HLS tools allow further and better tuning than handwritten code.
- Still, the code must be rearranged to maximize performance.