# Hardware deceleration

## The challenges of speeding up software

Intel Programmable Solutions Group

# Hardware acceleration - what is the challenge?

FPGA and SoC FPGA devices can be used as hardware accelerators to software

- Improve performance of calculations
- Improve power efficiency
- Can often be programmable to adapt to workloads at runtime

**How do we ensure the right architecture to achieve better performance?**
**What are the pitfalls if we get the architecture wrong?**

If we get the architecture wrong, then we could end up with
Hardware Deceleration!

# Before trying to accelerate with hardware….

Let’s ensure the software is optimized first

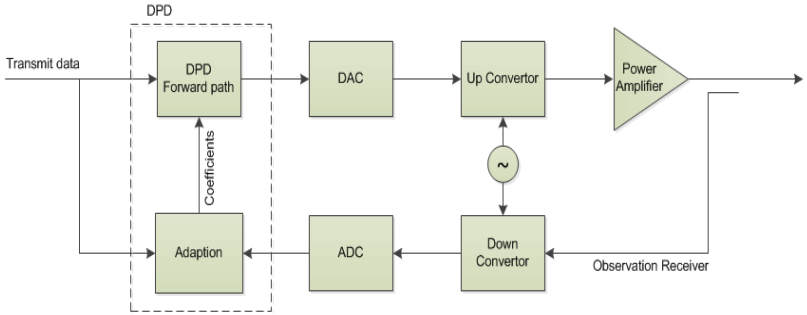
# Example - DPD - Digital Predistortion

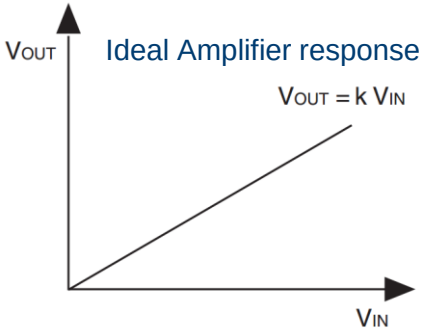
Power amplifiers (PA) for communications are not linear in their output

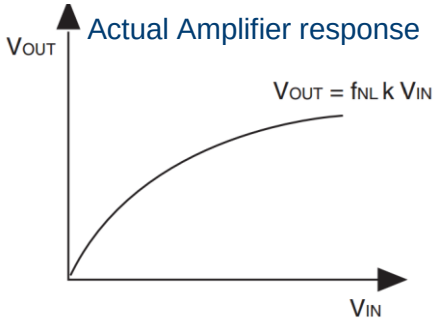
This non-linearity needs to be compensated for by pre-distorting the input signal to be broadcast.

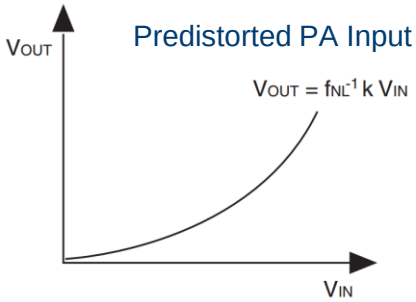
Digital Predistortion is the technology that implements this

This distortion changes with temperature and aging of the PA, so predistortion needs to adapt during operation

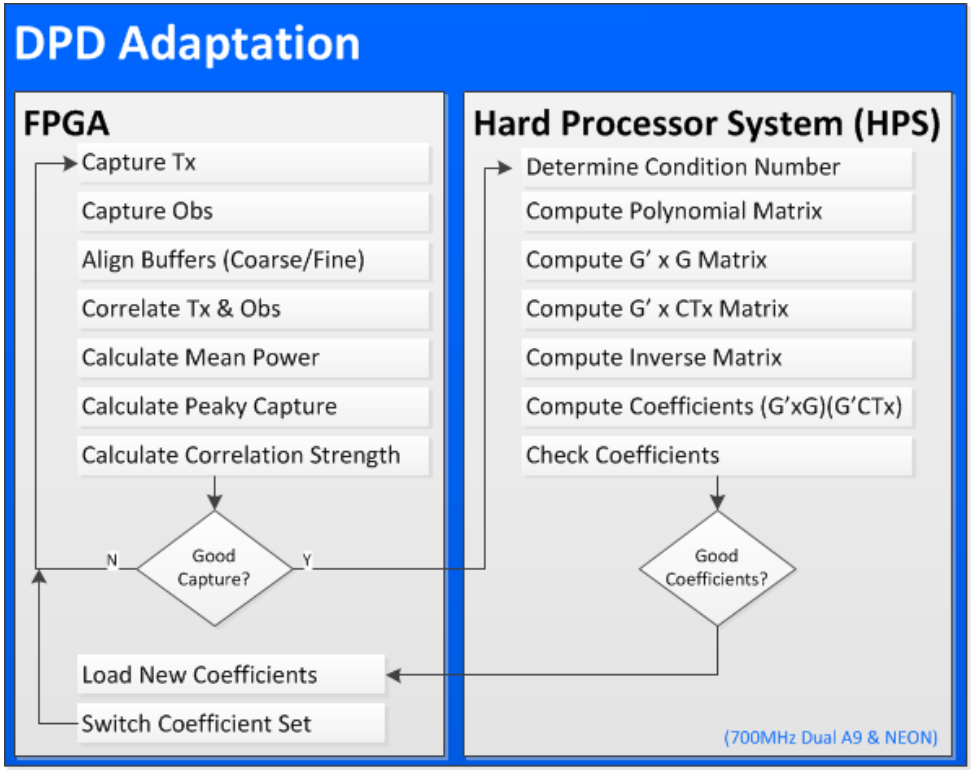


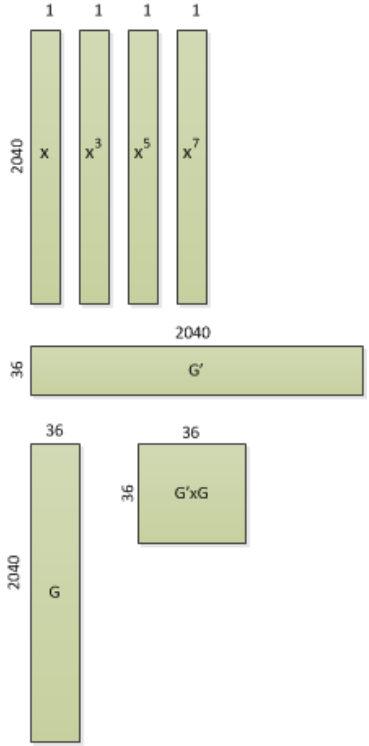



# DPD implementation



# DPD functions

CovarMatrix() G[36x2040] = CoRx[2048x1]

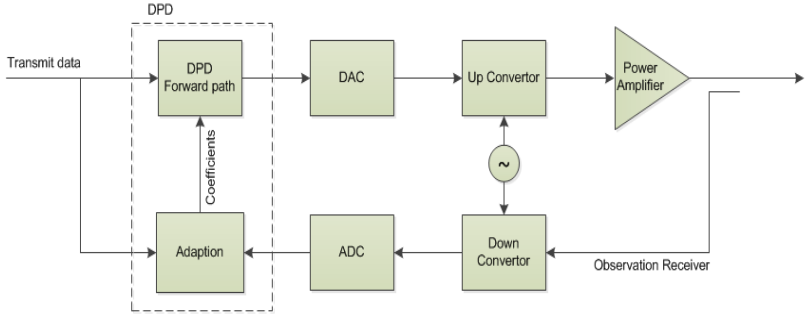
MM1_Gt_x_G()[36x36] = G'[36x2040] x G[2040x36]

MM2_Gt_x_CTx()[36x1] = G'[36x2040] x CTx[2048x1]

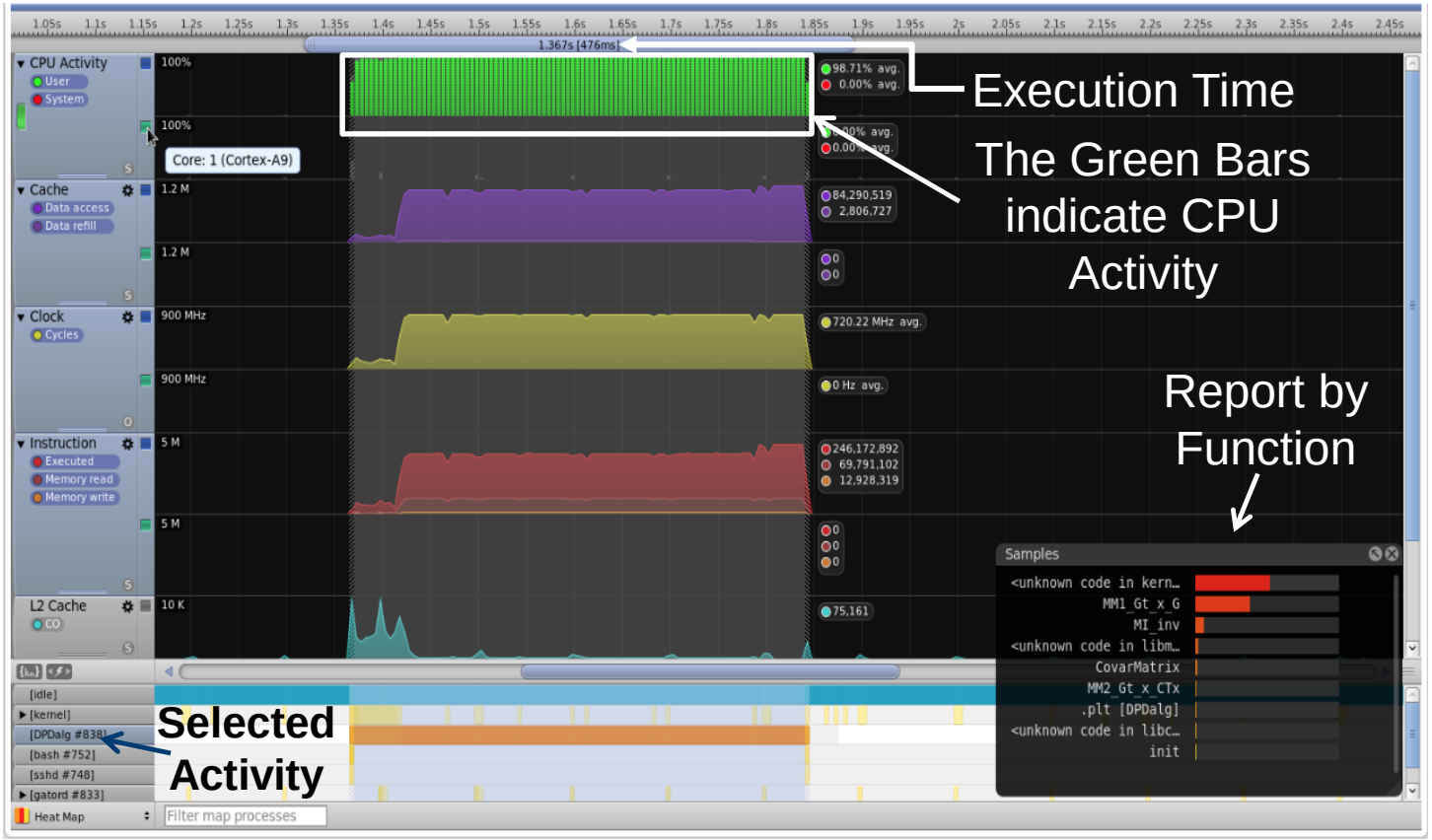
MI_inv() [36x36] = INV(MM1_Gt_x_G[36x36] )

MI_x_MM2()[36x1] Coeffs = MM1_Gt_x_G()[36x36] x MM2_Gt_x_CTx()[36x1]

$$\text{Coefficients} = \text{INV}(G' \times G) \times (G' \times CTx)$$



# Software profiling



# Call paths

| Self | % Self | Process | % Process | Total | Stack | [Process]/{Thread}/Code | Location |
|---|---|---|---|---|---|---|---|
| | | 7,649 | 100.00% | 94.13% | 0 | [idle] | - |
| | | 7,649 | 100.00% | 94.13% | 0 | {swapper/0 #0} | - |
| 7,649 | 100.00% | 7,649 | 100.00% | 94.13% | 0 | <unknown code in kernel> | <anonymous> |
| | | 467 | 100.00% | 5.75% | 0 | [DPDalg #838] | - |
| | | 467 | 100.00% | 5.75% | 0 | {DPDalg #838} | - |
| 359 | 76.87% | 359 | 76.87% | 4.42% | 160 | MM1_Gt_x_G | DPDalg.c:84 |
| 54 | 11.56% | 54 | 11.56% | 0.66% | 288 | MI_inv | DPDalg.c:151 |
| 19 | 4.07% | 19 | 4.07% | 0.23% | 0 | <unknown code in libm-2.15.so> | <anonymous> |
| 14 | 3.00% | 14 | 3.00% | 0.17% | 416 | CovarMatrix | DPDalg.c:50 |
| 9 | 1.93% | 9 | 1.93% | 0.11% | 160 | MM2_Gt_x_CTx | DPDalg.c:108 |
| 9 | 1.93% | 9 | 1.93% | 0.11% | 0 | <unknown code in kernel> | <anonymous> |
| 1 | 0.21% | 1 | 0.21% | 0.01% | 96 | init | DPDalg.c:27 |
| 1 | 0.21% | 1 | 0.21% | 0.01% | 0 | .plt [DPDalg] | DPDalg |
| 1 | 0.21% | 1 | 0.21% | 0.01% | 0 | <unknown code in libc-2.15.so> | <anonymous> |
| | | 8 | 100.00% | 0.10% | 0 | [kernel] | - |
| | | 1 | 100.00% | 0.01% | 0 | [bash #752] | - |
| | | 1 | 100.00% | 0.01% | 0 | [sshd #748] | - |
| | | 0 | 0.00% | 0.00% | 0 | [gatord #833] | - |
| | | 0 | 0.00% | 0.00% | 0 | [lighttpd #685] | - |

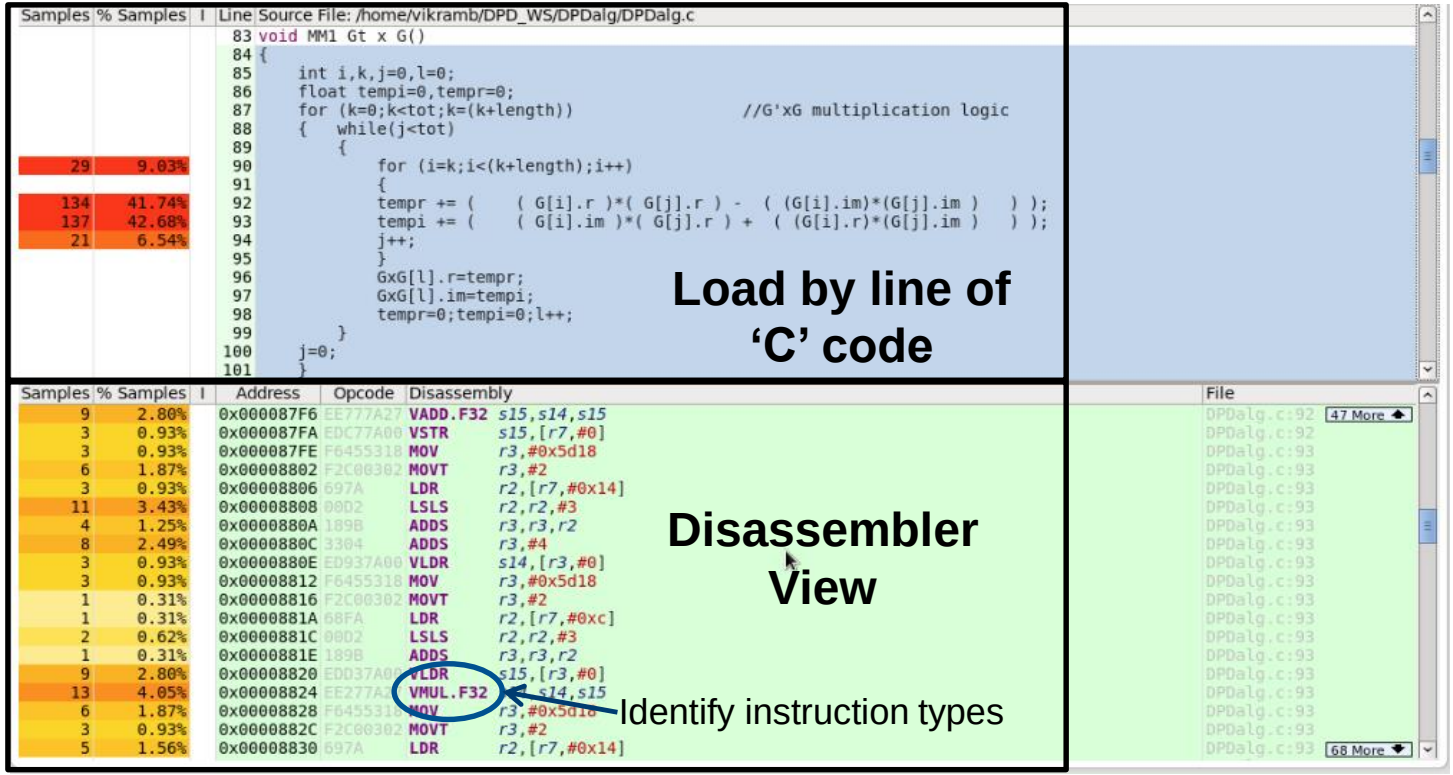
| Samples | % Samples | Instances | Function Name | Location |
|---|---|---|---|---|
| 359 | 76.87% | 1 | MM1_Gt_x_G | DPDalg.c:84 |
| 54 | 11.56% | 1 | MI_inv | DPDalg.c:151 |
| 19 | 4.07% | 1 | <unknown code in libm-2.15.so> | <anonymous> |
| 14 | 3.00% | 1 | CovarMatrix | DPDalg.c:50 |
| 9 | 1.93% | 1 | MM2_Gt_x_CTx | DPDalg.c:108 |
| 9 | 1.93% | 1 | <unknown code in kernel> | <anonymous> |
| 1 | 0.21% | 1 | init | DPDalg.c:27 |
| 1 | 0.21% | 1 | .plt [DPDalg] | DPDalg |
| 1 | 0.21% | 1 | <unknown code in libc-2.15.so> | <anonymous> |

**Idle Time**

**Selected Function**

**Function wise report**

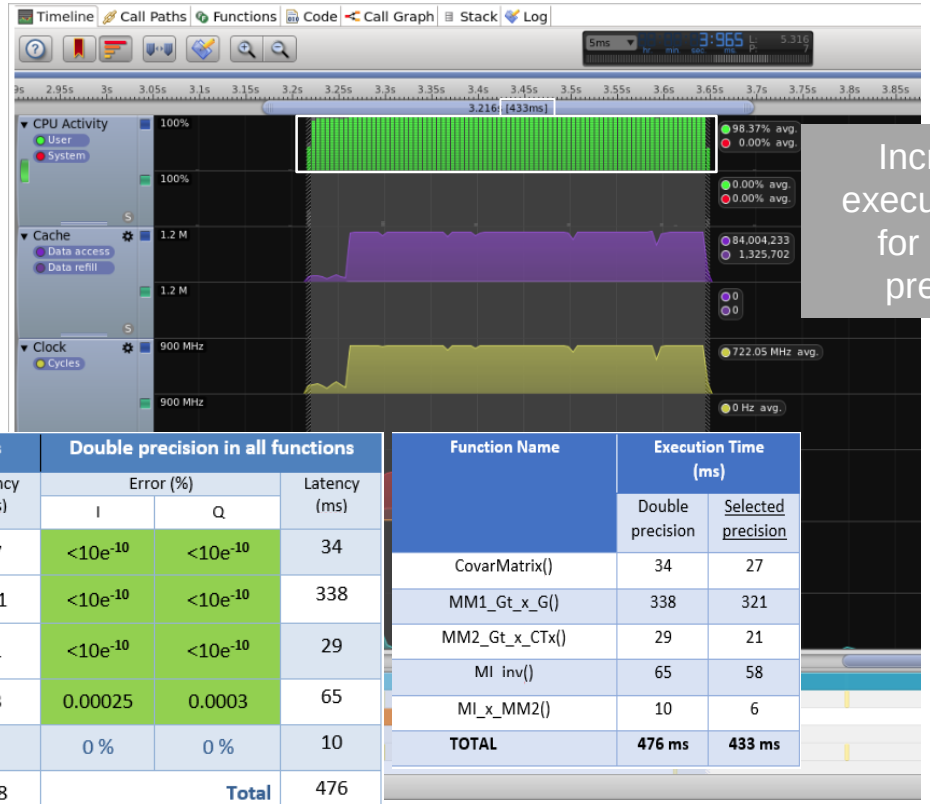
# Code view


Samples | % Samples | I | Line | Source File: /home/vikramb/DPD_WS/DPDalg/DPDalg.c
83 void MM1 Gt x G()
84 {
85 int i,k,j=0,l=0;
86 float tempi=0,tempr=0;
87 for (k=0;k<tot;k=(k+length)) //G'xG multiplication logic
88 { while(j<tot)
89 {
29 9.03% 90 for (i=k;i<(k+length);i++)
91 {
134 41.74% 92 tempr += ( ( G[i].r )*( G[j].r ) - ( (G[i].im)*(G[j].im ) ) );
137 42.68% 93 tempi += ( ( G[i].im )*( G[j].r ) + ( (G[i].r)*(G[j].im ) ) );
21 6.54% 94 j++;
95 }
96 GxG[l].r=tempr;
97 GxG[l].im=tempi;
98 tempr=0;tempi=0;l++;
99 }
100 j=0;
101 }
Load by line of 'C' code
Samples | % Samples | I | Address | Opcode | Disassembly | File
9 2.80% 0x000087F6 EE777A27 VADD.F32 s15,s14,s15 DPDalg.c:92 47 More
3 0.93% 0x000087FA EDC77A00 VSTR s15,[r7,#0] DPDalg.c:92
3 0.93% 0x000087FE F6455318 MOV r3,#0x5d18 DPDalg.c:93
6 1.87% 0x00008802 F2C00302 MOVT r3,#2 DPDalg.c:93
3 0.93% 0x00008806 697A LDR r2,[r7,#0x14] DPDalg.c:93
11 3.43% 0x00008808 00D2 LSLS r2,r2,#3 DPDalg.c:93
4 1.25% 0x0000880A 189B ADDS r3,r3,r2 DPDalg.c:93
8 2.49% 0x0000880C 3304 ADDS r3,#4 DPDalg.c:93
3 0.93% 0x0000880E ED937A00 VLDR s14,[r3,#0] DPDalg.c:93
3 0.93% 0x00008812 F6455318 MOV r3,#0x5d18 DPDalg.c:93
1 0.31% 0x00008816 F2C00302 MOVT r3,#2 DPDalg.c:93
1 0.31% 0x0000881A 68FA LDR r2,[r7,#0xc] DPDalg.c:93
2 0.62% 0x0000881C 00D2 LSLS r2,r2,#3 DPDalg.c:93
1 0.31% 0x0000881E 189B ADDS r3,r3,r2 DPDalg.c:93
9 2.80% 0x00008820 EDD37A00 VLDR s15,[r3,#0] DPDalg.c:93
13 4.05% 0x00008824 EE277A [illegible] s14,s15 DPDalg.c:93
6 1.87% 0x00008828 F6455318 MOV r3,#0x5d18 DPDalg.c:93
3 0.93% 0x0000882C F2C00302 MOVT r3,#2 DPDalg.c:93
5 1.56% 0x00008830 697A LDR r2,[r7,#0x14] DPDalg.c:93 68 More
Disassembler View
Identify instruction types

# Step 1: Latency vs. precision

Comparison of full floating point to single & double-precision

Determine Error (%) and latency trade-offs.




Increased execution time for double precision

| Functions | Single precision in all functions | | | Double precision in all functions | | |
|---|---|---|---|---|---|---|
| | Error (%) | | Latency (ms) | Error (%) | | Latency (ms) |
| | I | Q | | I | Q | |
| **CovarMatrix()** | <10e$^{-05}$ | <10e$^{-05}$ | 27 | <10e$^{-10}$ | <10e$^{-10}$ | 34 |
| **MM1_Gt_x_G()** | <10e$^{-05}$ | <10e$^{-05}$ | 321 | <10e$^{-10}$ | <10e$^{-10}$ | 338 |
| **MM2_Gt_x_CTx()** | <10e$^{-05}$ | <10e$^{-05}$ | 21 | <10e$^{-10}$ | <10e$^{-10}$ | 29 |
| **MI inv()** | 0.0435 | 0.019 | 53 | 0.00025 | 0.0003 | 65 |
| **MI_x_MM2()** | 0% | 0% | 6 | 0 % | 0 % | 10 |
| | | **Total** | 428 | | **Total** | 476 |

| Function Name | Execution Time (ms) | |
|---|---|---|
| | Double precision | Selected precision |
| CovarMatrix() | 34 | 27 |
| MM1_Gt_x_G() | 338 | 321 |
| MM2_Gt_x_CTx() | 29 | 21 |
| MI inv() | 65 | 58 |
| MI_x_MM2() | 10 | 6 |
| **TOTAL** | **476 ms** | **433 ms** |

# Step 2: Algorithmic optimization

A matrix multiplied with its transpose results is a square symmetric matrix

This property helps us reduce almost half the number of complex multiplications

All the elements in the lower triangular matrix can be obtained from upper triangular matrix

| Function Name | Execution time (ms) | |
|---|---|---|
| | Before algorithmic optimization | After algorithmic optimization |
| CovarMatrix() | 27 | 16 |
| MM1_Gt_x_G() | 321 | 262 |
| MM2_Gt_x_CTx() | 21 | 21 |
| MI_inv() | 58 | 55 |
| MI_x_MM2() | 6 | 6 |
| **Total** | **433 ms** | **360 ms** |

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix}.$$

# Optimized code

## Original Pseudo Code

```
for (k=0;k<tot;k(k+length))
   for(i=k;i<(k+length);i++)
{  tempr +=((G[i].r)*(G[j].r) – (G[i].im)*(G[j].im));
   tempi +=((G[i].im)*(G[j].r) + (G[i].r)*(G[j].r));
```

134|42%

137|42%

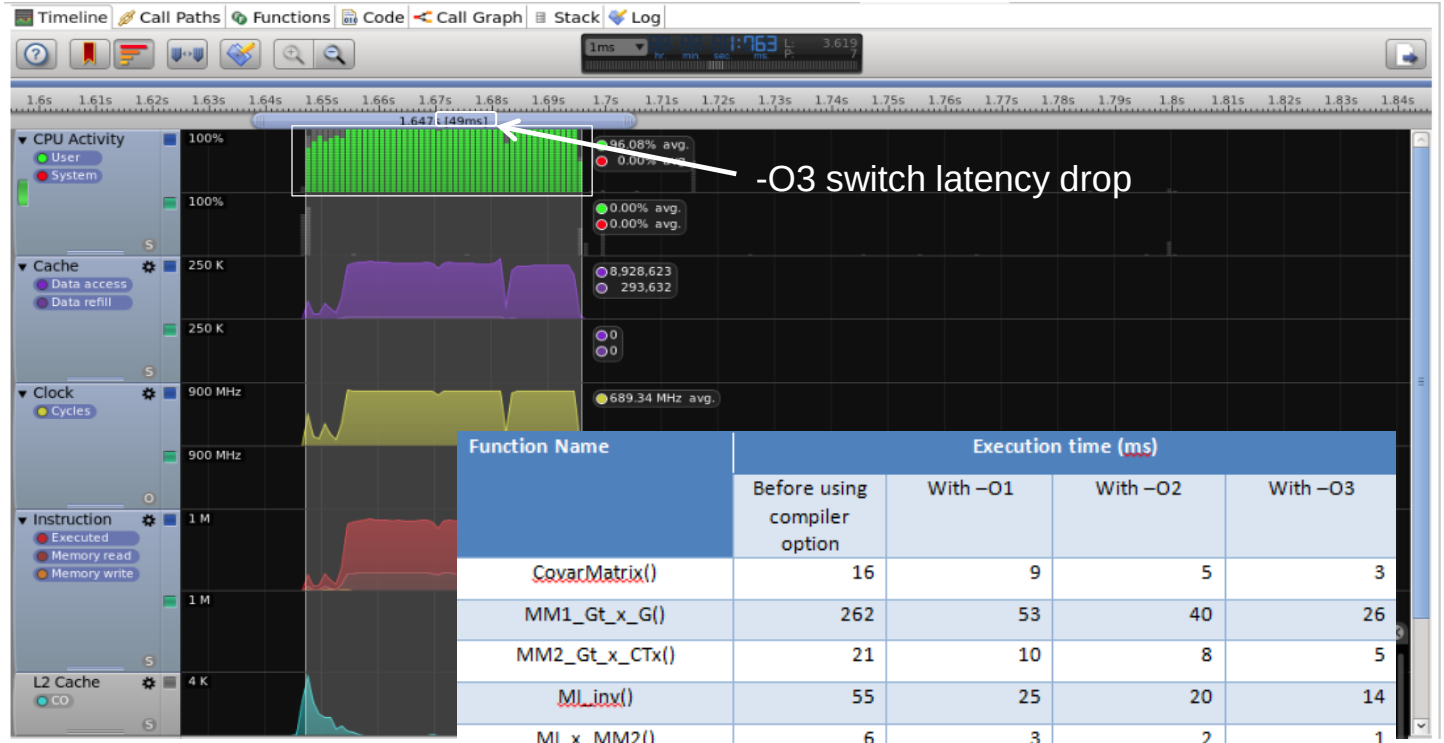
## Optimized Code

```
for (h=0;h<MD;h++)
   for(i=0;i<4;i++)
      for(k=0;l<MD;k++)
         for(l=0;l<4;l++)
            for(m=(MD-k-1);m<(corxlength-k);m++)
tempr += (((G+j+(i*corxlength))-> r*(G+m+(l*corxlength))->r) –
          ((G+j+(i*corxlength))->im*(G*m+(l*corxlength))->r));
tempi += (((G+j+(i*corxlength))->im*(G+m+(l*corxlength))->r) +
          ((G+j+(i*corxlength))-> r*(G*m+(l*corxlength))->r));
```

105|37%

115|39%

# Step 3: Compiler options – OPT-3




| Function Name | Execution time (ms) | | | |
|---|---|---|---|---|
| | Before using compiler option | With –O1 | With –O2 | With –O3 |
| CovarMatrix() | 16 | 9 | 5 | 3 |
| MM1_Gt_x_G() | 262 | 53 | 40 | 26 |
| MM2_Gt_x_CTx() | 21 | 10 | 8 | 5 |
| MI_inv() | 55 | 25 | 20 | 14 |
| MI_x_MM2() | 6 | 3 | 2 | 1 |
| **Total Time** | **360ms** | **99ms** | **75ms** | **49ms** |

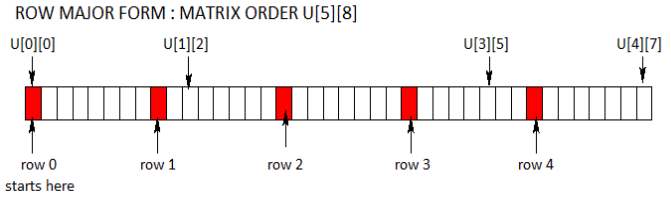
# Step 4: NEON* vectorization

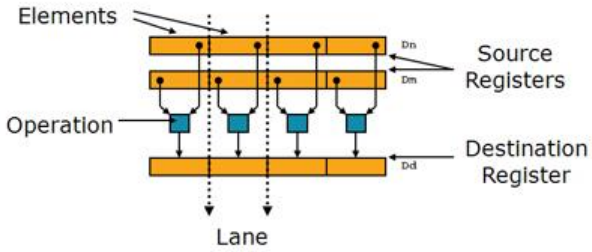
## Data types

- signed/unsigned 8-bit, 16-bit, 32-bit, single-precision

## Code Changes

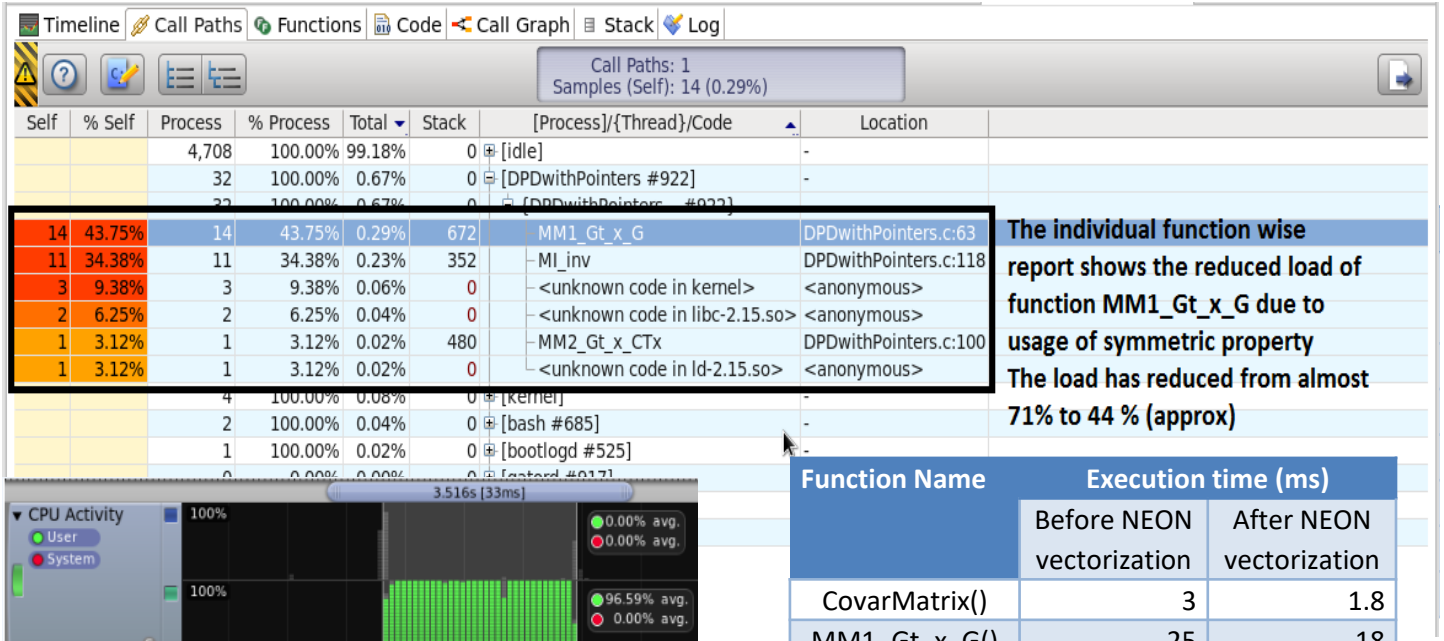
- Use of 1D array instead of 2D array
- Use of keyword ‘restrict’ for unaligned pointers
- Use of ‘const *’ for aligned pointers




| Description | Resource | Path | Location |
|---|---|---|---|
| Infos (6 items) | | | |
| vectorized 0 loops in function. | DPDalg.c | /DPDalg | line 26 |
| vectorized 1 loops in function. | DPDalg.c | /DPDalg | line 83 |
| vectorized 1 loops in function. | DPDalg.c | /DPDalg | line 107 |
| vectorized 1 loops in function. | DPDalg.c | /DPDalg | line 128 |
| vectorized 2 loops in function. | DPDalg.c | /DPDalg | line 49 |
| vectorized 4 loops in function. | DPDalg.c | /DPDalg | line 150 |

# Vectorization




**The individual function wise report shows the reduced load of function MM1_Gt_x_G due to usage of symmetric property The load has reduced from almost 71% to 44 % (approx)**

| Function Name | Execution time (ms) | |
|---|---|---|
| | Before NEON vectorization | After NEON vectorization |
| CovarMatrix() | 3 | 1.8 |
| MM1_Gt_x_G() | 25 | 18 |
| MM2_Gt_x_CTx( ) | 6 | 4 |
| MI_inv() | 13 | 9.3 |
| MI_x_MM2() | 1 | <1 |
| **Total** | **48 ms** | **33 ms** |

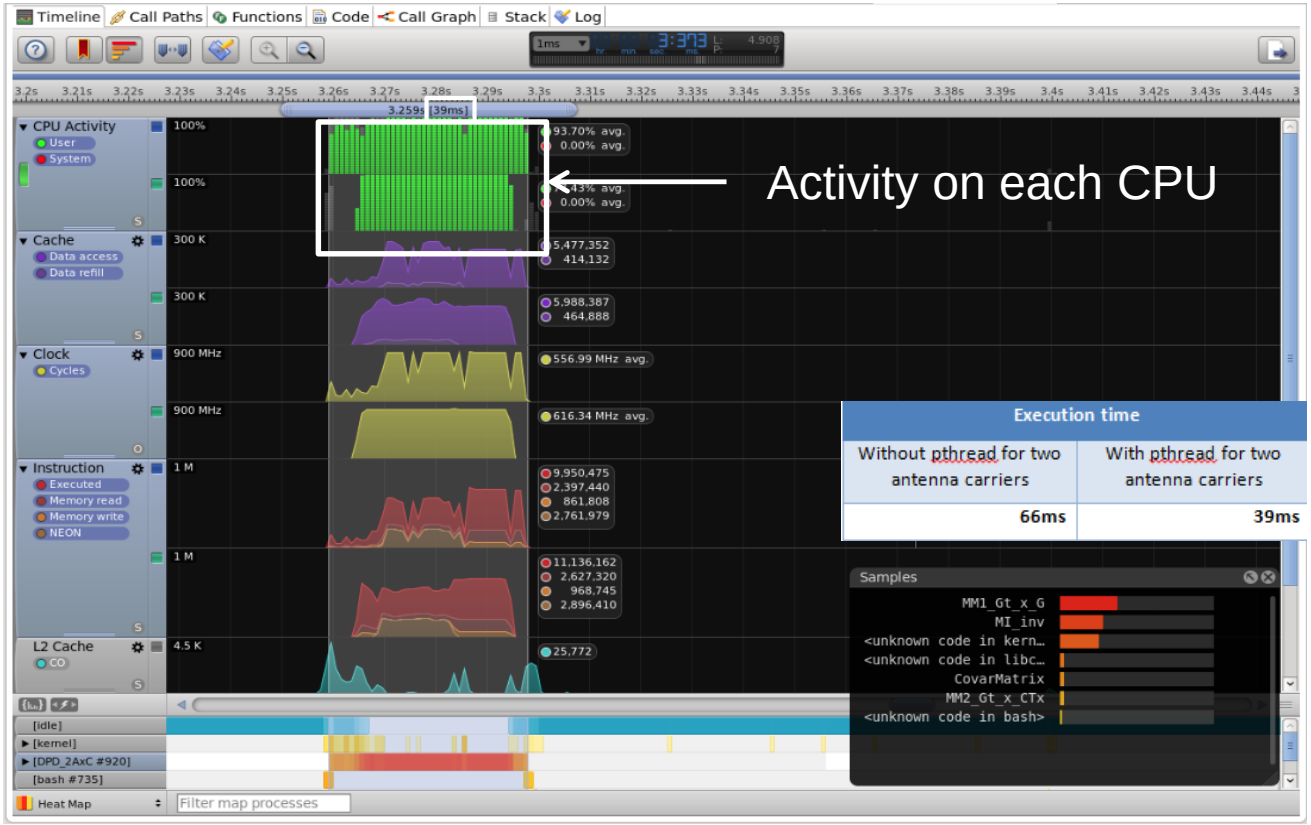
# Step 5: Multi-threading

```
#include <pthread.h>
#include <stdio.h>

pthread_t thread1, thread2;
int main()
{
        int rc;
        rc = pthread_create(&thread1, NULL, dpd_algorithm,(void*)&dpd1);  //Ax 1
        if (rc) {
                exit(-1);
        }
        rc = pthread_create(&thread2, NULL, dpd_algorithm,(void*)&dpd2);  //Ax 2
        if (rc) {
                exit(-1);
        }
        pthread_exit(NULL);
        return 0;
}
```

| Self | % Self | Process | % Process | Total | Stack | [Process]/{Thread}/Code | Location |
|---|---|---|---|---|---|---|---|
| | | 9,716 | 100.00% | 99.28% | 0 | [idle] | - |
| | | 63 | 100.00% | 0.64% | 0 | [DPD_2AxC #920] | - |
| | | 30 | 47.62% | 0.31% | 0 | {DPD_2AxC #921} | - |
| 14 | 22.22% | 14 | 22.22% | 0.14% | 640 | MM1_Gt_x_G | DPD_2AxC.c:85 |
| 11 | 17.46% | 11 | 17.46% | 0.11% | 320 | MI_inv | DPD_2AxC.c:156 |
| 2 | 3.17% | 2 | 3.17% | 0.02% | 0 | <unknown code in kernel> | <anonymous> |
| 1 | 1.59% | 1 | 1.59% | 0.01% | 128 | CovarMatrix | DPD_2AxC.c:60 |
| 1 | 1.59% | 1 | 1.59% | 0.01% | 480 | MM2_Gt_x_CTx | DPD_2AxC.c:135 |
| 1 | 1.59% | 1 | 1.59% | 0.01% | 0 | <unknown code in libc-2.15.so> | <anonymous> |
| | | 29 | 46.03% | 0.30% | 0 | {DPD_2AxC #922} | - |
| 15 | 23.81% | 15 | 23.81% | 0.15% | 640 | MM1_Gt_x_G | DPD_2AxC.c:85 |
| 11 | 17.46% | 11 | 17.46% | 0.11% | 320 | MI_inv | DPD_2AxC.c:156 |
| 1 | 1.59% | 1 | 1.59% | 0.01% | 128 | CovarMatrix | DPD_2AxC.c:60 |
| 1 | 1.59% | 1 | 1.59% | 0.01% | 480 | MM2_Gt_x_CTx | DPD_2AxC.c:135 |
| 1 | 1.59% | 1 | 1.59% | 0.01% | 0 | <unknown code in libc-2.15.so> | <anonymous> |

# Multi-threading



# Summary of SW optimisation techniques

1. Compiler settings
2. Algorithmic optimisation (code architecture modification)
3. Data flow optimisation (code architecture modification)
4. Target processor special hardware (e.g. NEON framework vectorization)
5. Implement Multi-threading/multi-core

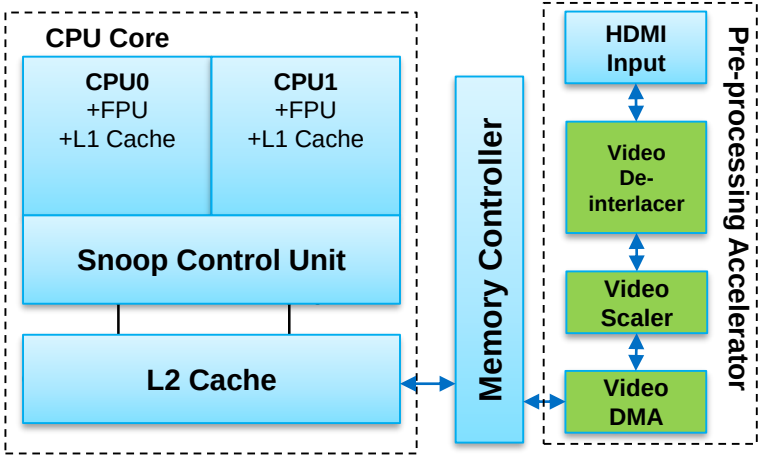
# Acceleration technique #1:

Bump in the wire / pre-processing / post-processing

# Bump in the wire techniques

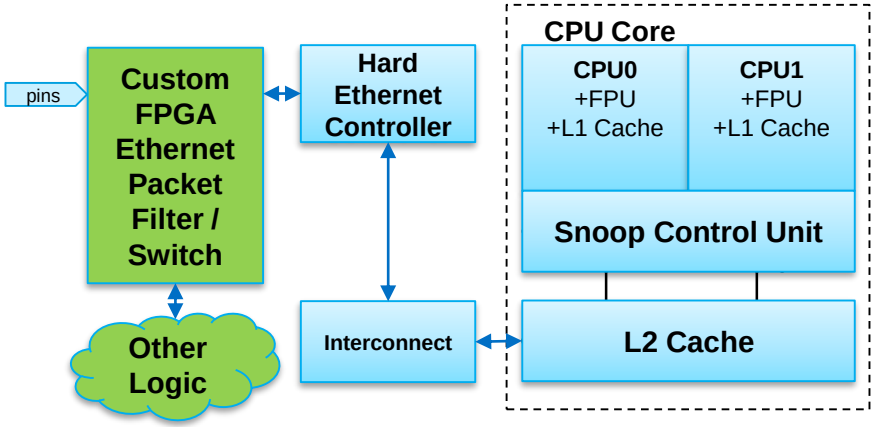
## Pre-processing / Post-processing

- Data will be transformed prior / after arrival at the CPU System




## Bump in the wire

- The data will be filtered / worked on before arriving at the original interface and can be worked on after data exits the processor



# Bump in the wire - potential for deceleration

Pre- or post-processing of data could ….

**Add to the latency between the processor and the I/O**.

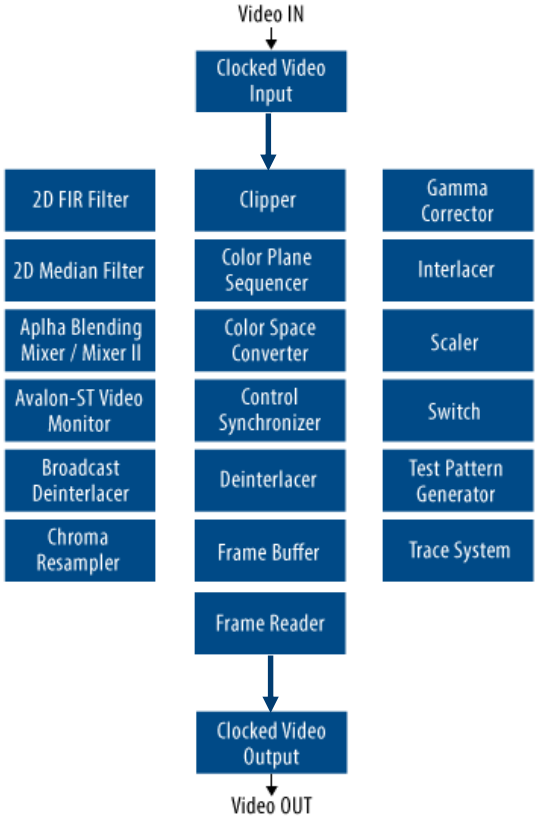
In some instances, the latency of the transaction is critically important to the overall system performance, and needs to be minimized.

# Bump in the wire - potential for acceleration

Performance gains should be relatively straight forward to estimate.

CPU offload can significantly speed up memory-intensive functions or reduce CPU utilization

The FPGA is real-time tasks performed with clock-cycle accuracy / determinism.



# Acceleration technique #2:

## Tightly-coupled instruction set extension

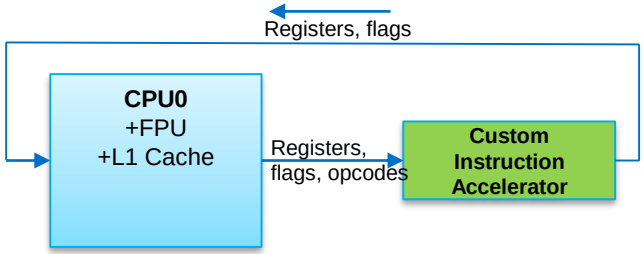
# Tightly-coupled instruction set extension

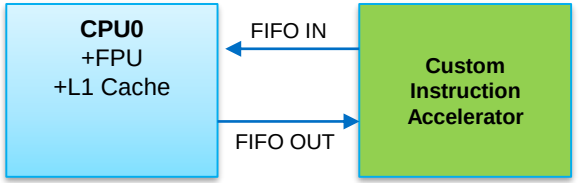
Custom instructions are used to extend the operation of the soft processor

- Data is passed via registers, flags, opcodes to acceleration core from processor

**or**

- Data is passed via FIFO interface with flags to acceleration core from processor




*Note: CI accelerators can have their own local memory*

# Tightly-coupled instruction set extension - potential for deceleration

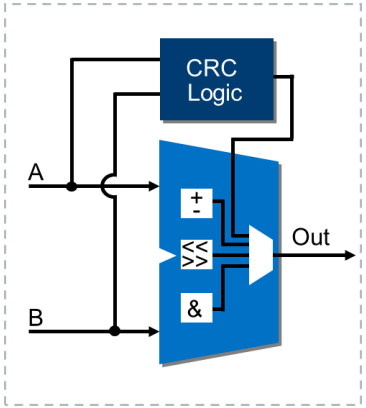
Tightly-coupled instruction set extension could reduce performance by…

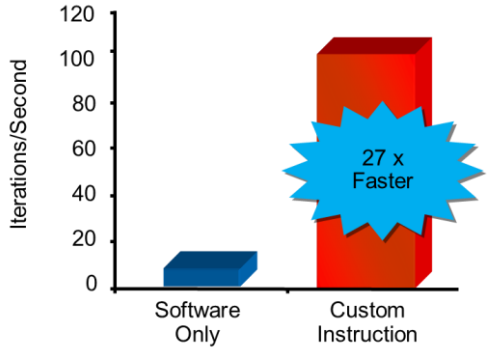
**Adding bubbles to the CPU pipeline if the instruction is multi-cycle**
**Slow down the FPGA if timing is hard to close**
**Halt the processor for many cycles if data is fetched externally**

In architectures where the custom instruction can cause a CPU stall, it is important to ensure that efficiency of the CPU pipeline is maintained. If the custom instruction has a dependency on an external data source it could stall for a long time waiting for a new data value.

# Tightly-coupled instruction set extension - potential for acceleration

If the accelerated function would be used frequently by critical code in the system, then an acceleration can be achieved. Tight code loops can be accelerated compared to a multi-instruction approach due to the close proximity of the accelerator to the CPU (In or close to the instruction pipeline).




| Function | Operation | Cycles to Execute |
|---|---|---|
| Arithmetic | add/subtract<br>multiply<br>divide<br>negate/absolute | 5<br>4<br>16<br>1 |
| Roots | square root | 8 |
| Conversions | integer to float<br>float to integer | 4<br>2 |
| Comparisons | min/max<br>less than/equal<br>greater than/equal<br>equal/not equal | 1<br>1<br>1<br>1 |

Example Floating point offload performance

CRC algorithm design example: https://www.altera.com/support/support-resources/design-examples/intellectual-property/embedded/nios-ii/exm-custom-instruction.html
Floating point Cycle counts as documented in Intel Platform Designer version 17.1

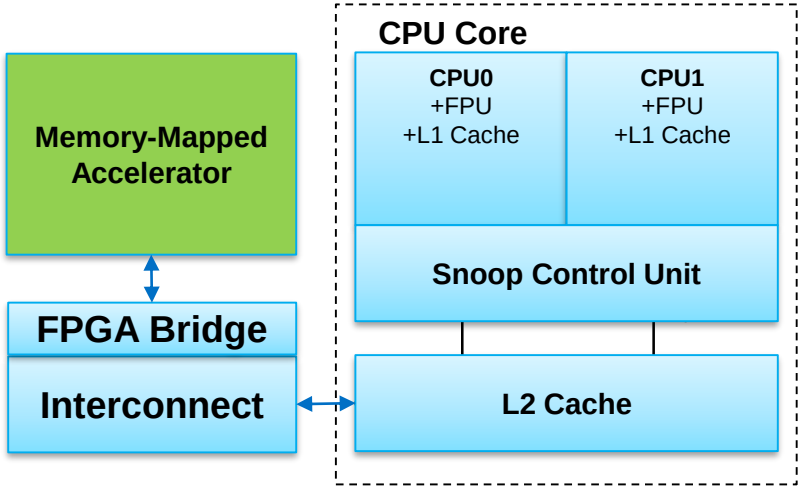
# Acceleration technique #3:

## Tightly-coupled memory-mapped accelerator

# Tightly-coupled memory-mapped accelerator

Accelerator is created on-chip using FPGA resources

- Access to the accelerator is via on-chip bus - Eg. AXI, Avalon
- Parallel, high-bandwidth access is available from processor system to accelerator for read/write access



# Tightly-coupled memory-mapped accelerator - potential for deceleration

A tightly-coupled memory-mapped accelerator could reduce performance by…

**Being a victim of increased latency**

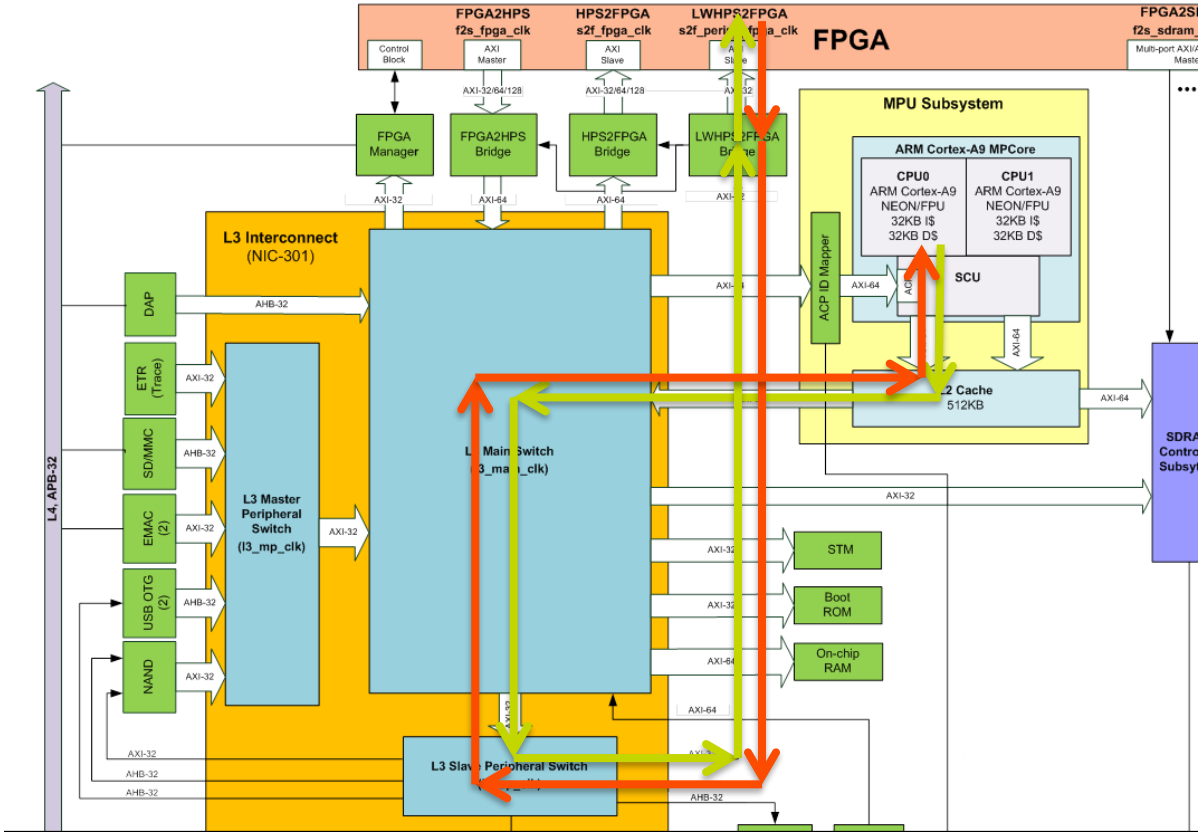
- If the control loop is tight, then the latency can kill performance
- Latency from a SoC hard processor to FPGA can be tens of CPU clock cycles

# How do we actually measure latency?

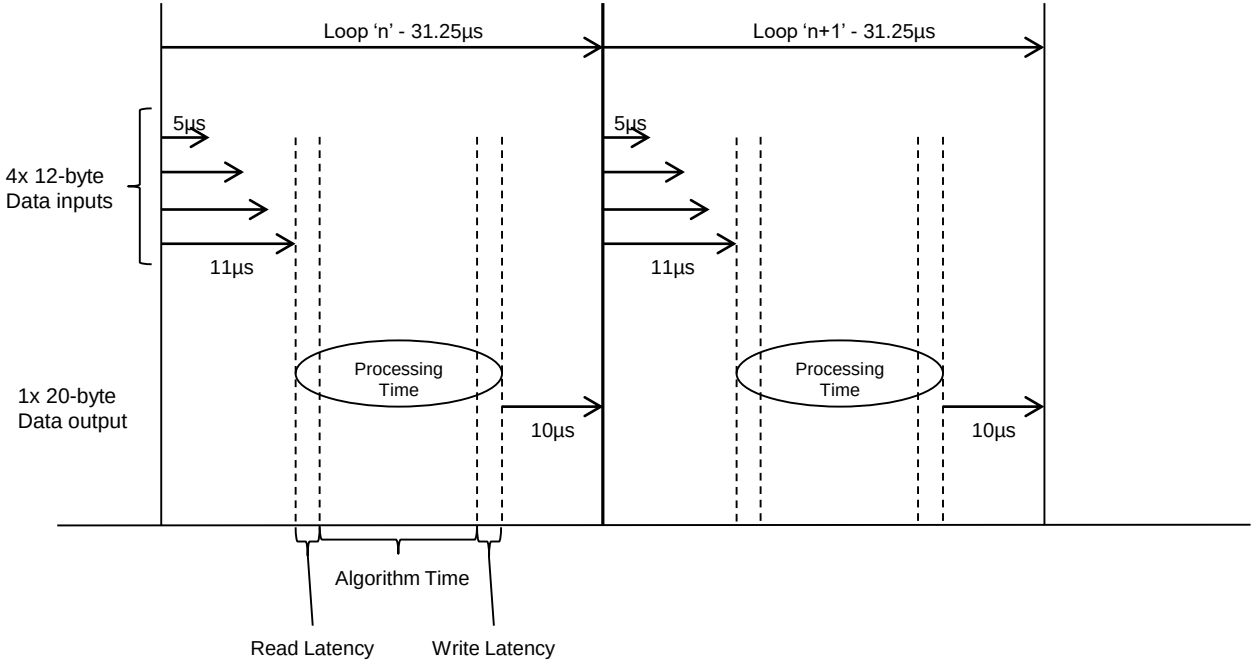
Example scenario:

- What is the latency of a 12-byte read?
- What is the latency of a 48-byte read?
- What is the latency of a 20-byte write?

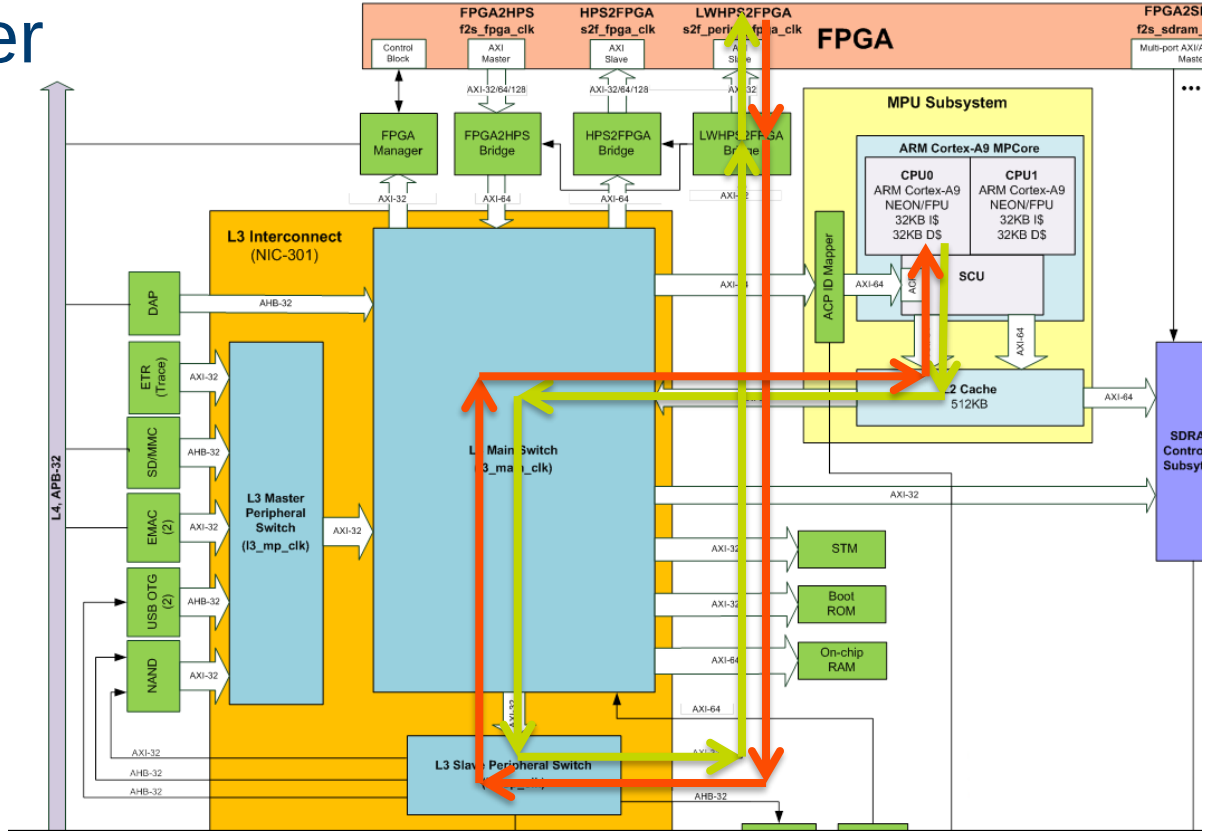
# The measurement challenge



# Clarification - what are you trying to do?



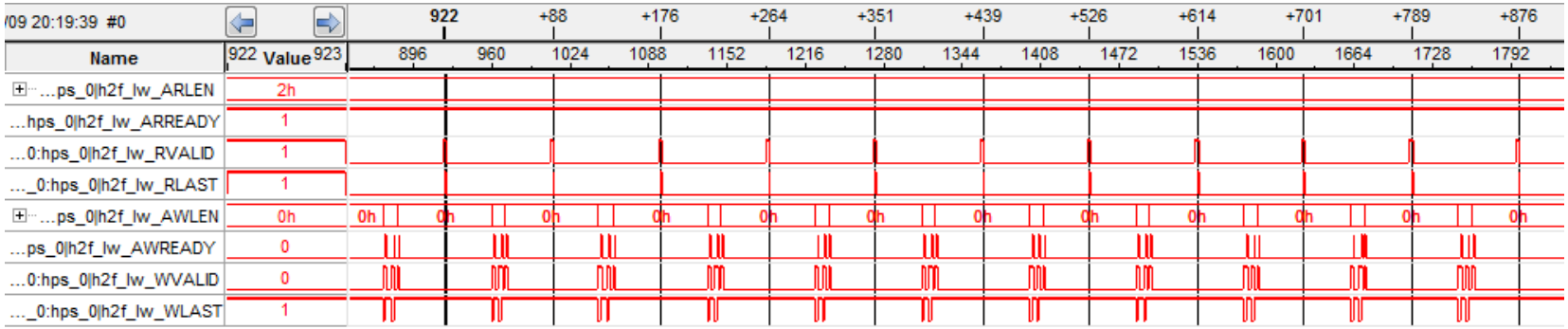
# So really, we can benchmark a read and write together



# Example latency measurement technique

- Software debugger can timestamp cycle counts for a series of transactions
- Hardware or FPGA logic analyzer can measure period between loops of transactions

| Cycles | | Detail |
|---|---|---|
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 568 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 568 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 560 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 568 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 568 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 560 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |
| 568 | | LDM r0,{r1-r3} |
| 0 | | STM r0,{r1-r5} |
| 0 | | B asm_test3 ; 0x10002F4 |



# Tightly-coupled memory-mapped accelerator - potential for acceleration

- Streaming data fits well with memory-mapped accelerators
- Best performance if results can be read separately to data coming in - the CPU can work on other processing in the mean-time
- DMA can further offload processor from reading directly from peripheral

# DPD example - FPGA acceleration

## MM1_Gt_x_G() calculation consumes 50% of the entire processing time

- 18ms/39ms (2 antennas)

## FPGA acceleration can reduce this time to **<<0.1ms**

- Also exploit matrix shifts (rows are delayed versions)

## Total Latency 22ms

- for 2 antennas on low-cost FPGA

# DPD Example - Optimization Summary

| | Cores used (Cortex A9) | Optimization | Time (ms) per Antenna |
|---|---|---|---|
| Software Optimization | 1 | OPT-0 (Vanilla implementation) | 433 |
| | 1 | OPT-0 (algorithmic optimization) | 360 |
| | 1 | OPT-1 | 99 |
| | 1 | OPT-2 | 75 |
| | 1 | OPT-3 | 48 |
| | 1 | OPT-3 + NEON Vectorization | 33 |
| | 2 | OPT-3 + NEON + pthread (for two antennas) | 39/2 |
| Hardware acceleration | 2 | OPT-3 + NEON + pthread (2 antennas) + FPGA Acceleration | 21/2 |

***~2x FPGA HW Acceleration vs Optimised Software***

All numbers were measured on a DPD implementation targeting Intel Cyclone V SoC. More detail and full figures available at https://www.altera.com/en_US/pdfs/literature/wp/wp-01248-dpd-profiling-and-optimization-with-altera-socs.pdf

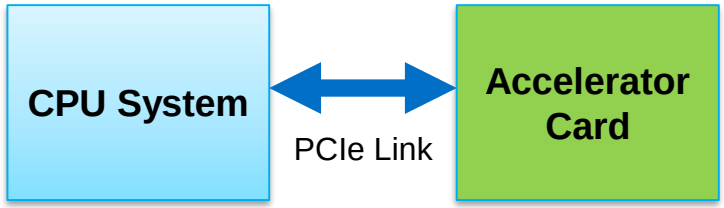
# Acceleration technique #4:

External bus-connected accelerator card

# External, bus-connected Accelerator card

CPU is connected to accelerator via external link, E.g. PCI Express*

- Accelerator is memory-mapped to CPU
- Interrupts also available across the link
- Multiple accelerators may be installed in the same system



# Accelerator card - potential for deceleration

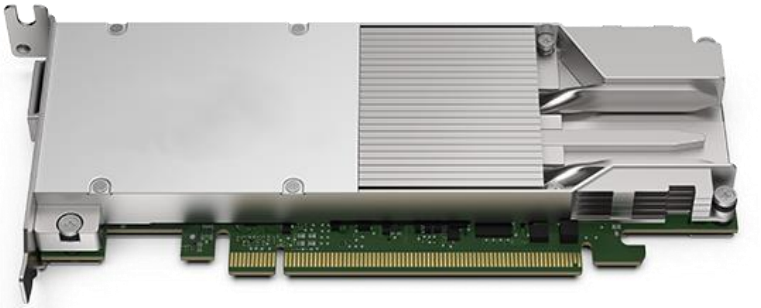
An accelerator card could reduce performance by…

**Being a victim of increased latency**

- If the control loop is tight, then the latency can kill performance
- Latency from a CPU to a PCIe* card can be much longer than on-chip interfaces

# External bus-connected accelerator card - potential for acceleration

Where data can be constantly streamed over the interface to the accelerator card, and resultant calculations returned, the latency of the link only serves as an initial latency, and once the pipeline is filled, full-bandwidth use can be made of the accelerator.



Database Acceleration Trial -

- 10x faster Real-time data analytics[1]
- 2x Traditional data warehousing[2]
- 3x Storage Compression[3]

Genomics sequencing Trial -

- 50x Pair HMM speedup[4]

1. Based on database queries run with SWARM64 acceleration vs. no acceleration. Testing performed by Swarm64. See System Configurations page for more details.
2. Data warehousing tested with queries and data taken from TPC-DS benchmark. Testing performed by Swarm64. See System Configurations for more details.
3. Based on database size run with SWARM64 acceleration vs. no acceleration. Testing performed by Swarm64. See System Configurations page for more details.
4. Based on tests by the Broad Institute. See System Configurations page for more details of Test configuration.

Tests measure performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance.
Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

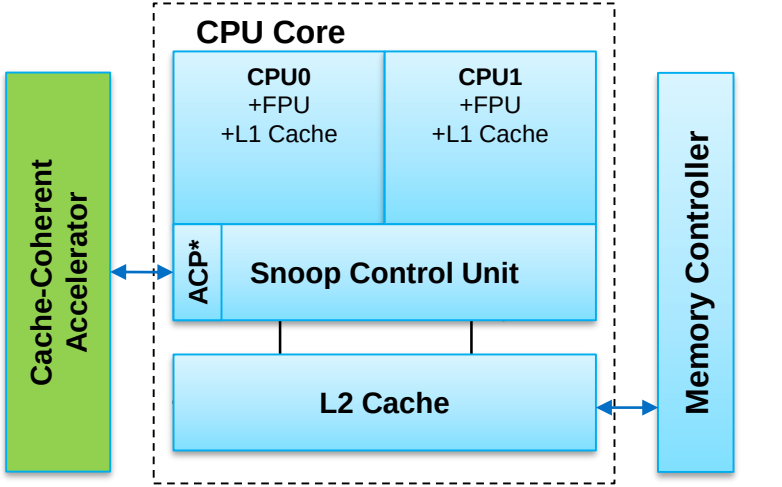
# Acceleration technique #5:

Cache-coherent accelerator with on-chip processor interface

# Cache-coherent accelerator with on-chip processor interface

Accelerator is created from FPGA on-chip resources

- Access for data transit is via ACP interface to cache-coherent port
- Transactions marked as cacheable will be directly sourced from cache as appropriate




*ACP = Accelerator Coherency Port

# Cache-coherent accelerator - potential for deceleration

An cache-coherent accelerator reduce performance by…

**Thrashing the caches**

- Large data transfers through cache interfaces will cause existing cache data to be swapped out
- Other CPU workloads could slow down if this data needs to be re-fetched
- Cache line locking can help, but has knock-on effects on performance

# Cache-coherent accelerator - potential for acceleration

Participation in cache-coherence can have major performance benefits in some instances compared to memory mapped systems.

- No need for CPU to flush data to external memory
- Direct hardware access to cached data without DDR interfacing
- Payload returned close to CPU, so long as it fits in the cache

# Virtualization:

The effects of virtualization on hardware performance

# Use of virtualization is increasingly popular

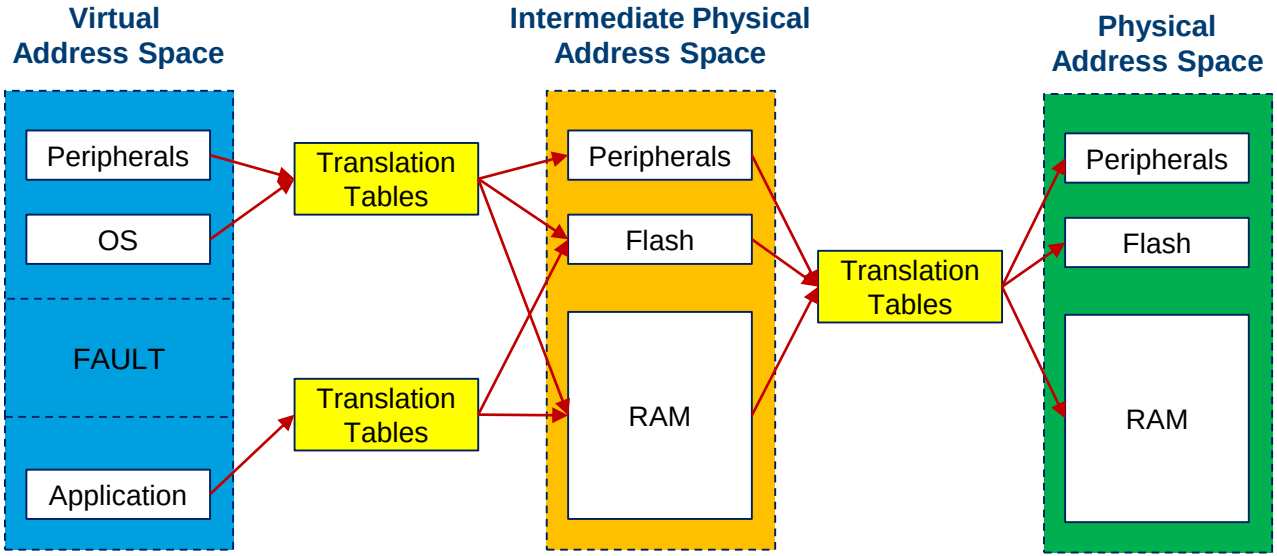
Multi-user environments

More than one O/S required to run legacy and new code

Hypervisors used to manage resources and permissions to hardware

Additional architectural complexity to support hypervisor architecture

# Virtualization - adds additional address decode for each virtual OS.



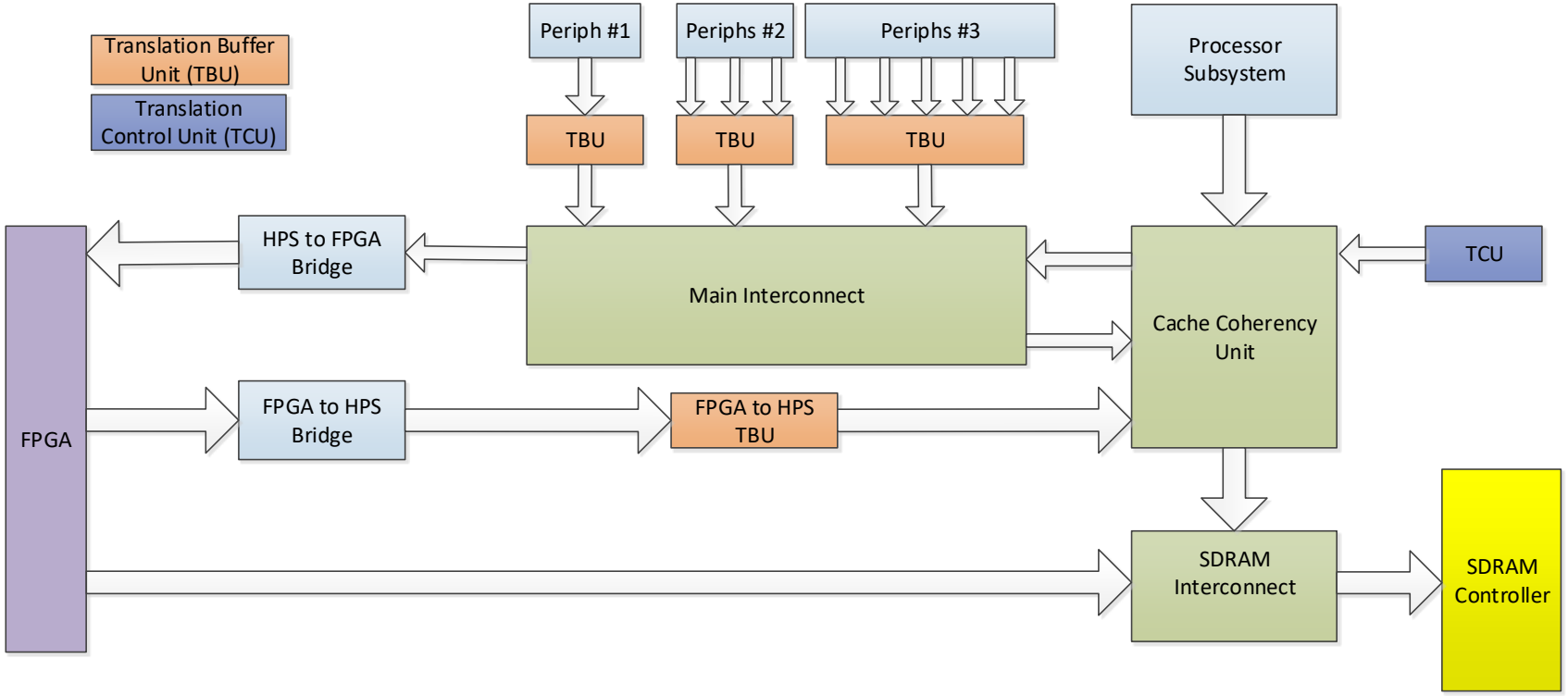
# Virtualization - potential for deceleration

Virtualization could decrease system performance by …

**Adding additional address decode logic**

- Architecture is key here.  Local caching of translation of translation buffers for the System MMU is key to maintain performance.

# Example System MMU infrastructure



# Virtualization - potential for acceleration

Virtualization could increase system performance by …

**Allowing multiple O/S to be hosted on one device**

- Off-chip latency has been seen to be critical in some designs. Memory can be safely shared, resources saved and systems optimized for data throughput.

# Summary

# Summary

1. Bump in the wire:
   Pros: Independent of processor & memory, deterministic, can be low latency
   Cons: Adds latency to arrival of data

2. Processor custom instructions
   Pros: local data, tight integration with software
   Cons: Can stall processor/block other processes

3. On-chip memory-mapped accelerator
   Pros: Can act independently of processor, processor can do other tasks, data can stream direct from peripheral
   Cons: Adds latency to arrival of data to processor, has to be on-chip

4. External bus-connected accelerator card
   Pros: Can act independently of processor, processor can do other tasks, data can stream direct from external interface, easy to implement, can support multiple instances
   Cons: Latency is high

5. Cache-coherent accelerator
   Pros: Local data, low latency, potential high performance
   Cons: Data set must be small and cache aligned, processor/data has to be managed to avoid cache thrash

# Conclusion

The performance improvement of a processor system with accelerators is greatly influenced by **architectural choices**.

The architect needs to consider **latency** and **data flow** dependencies to make sound implementation choices.  In that way pipeline bubbles and other effects that can negatively affect performance are minimized.

Swarm64 system configuration: Supermicro* SuperServer 2028U-TR4+ with Super X10DRU-i+ Mainboard, 2X Intel® Xeon® E5-2695 v4 CPUs, 8X Samsung* 32GB DDR4-2400 ECC RAM.
Note: This is SQL to relational database, not SQL to semi/unstructured data. Data Warehousing tested with data taken from the TPC-DS benchmarking suite and tested against the 99 TPC-DS query templates.

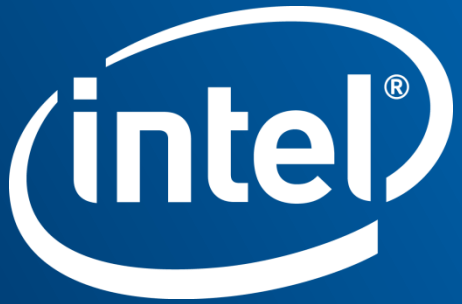
Broad Institute system configuration: Intel® Xeon® processor E5-2697 v2 @ 2.70 GHz, 2 sockets/12 cores per socket, 128 GB RAM, 2 TB Seagate HDD ST2000DM001 with Intel Arria 10 GX FPGA compared to Intel® Xeon® processor E5-2699 v4 @ 2.20 GHz, 2 sockets/22 cores per socket, 256 GB RAM, 2 TB Intel® SSD Data Center P3700 Series with single-core Intel® Advanced Vector Extensions (AVX) technology.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice Revision #20110804


intel®