ECE69500R System-On-Chip Design

Accelerating the Inference Process for MLP and CNN

Background:

The MLP and CNN are trained using the C based light weight KANN neural network library. Our goal was to use the pre-trained models from this library and accelerate the inference process for a set of 100 MNIST test images for both the networks while still maintaining high levels of accuracy in classification.

We have used the Intel Quartus Prime (programmable logic device design tool) and ALTERA DE2-115 FPGA board for the project implementation. The software runs on the NIOS IIf processor, and accesses the training models and test images from the flash memory. All the comparisons have been made with respect to the baseline MSIST-NN reference software already provided.

Profiling the Reference Software:

To get the overall picture of the base software and to identify the key components, which are compute-intensive operations and can be implemented as hardware, profiling was done using Gprof and Performance Counter.

- **GPROF Profiling Tool:**

Multi-Layer Perceptron (set of 100 images)

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
37.50	9.46	9.46	5177389	0.00	0.00	__mulsf3
24.13	15.55	6.09	5100396	0.00	0.00	__addsf3
21.13	20.87	5.33	7400	0.00	0.00	kad_sdot
7.65	22.80	1.93				alt_get_errno
1.43	23.16	0.36	1	0.36	1.55	kann_data_read
1.14	23.45	0.29	55208	0.00	0.00	__umodsi3
1.03	23.71	0.26	78400	0.00	0.00	_strtod_l
0.92	23.95	0.23	2425	0.00	0.00	memcpy
0.70	24.12	0.18	55208	0.00	0.00	__udivsi3
0.63	24.28	0.16	1	0.16	23.13	main

Convolutional Neural Network (set of 10 images)

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ks/call	total Ks/call	name
43.21	1130.31	1130.31	609523001	0.00	0.00	__mulsf3
29.45	1900.55	770.24	614982000	0.00	0.00	__addsf3
22.97	2501.32	600.77	5543400	0.00	0.00	kad_sdot
1.08	2529.54	28.22				alt_get_errno
0.95	2554.33	24.79	749000	0.00	0.00	kad_saxpy_inlined
0.94	2578.95	24.62	55336	0.00	0.00	__umodsi3
0.31	2587.15	8.20	206	0.00	0.01	kad_op_conv2d
0.23	2593.06	5.91	15568003	0.00	0.00	__clzsi2
0.18	2597.83	4.77	200	0.00	0.00	conv2d_move_1to3
0.18	2602.42	4.59	309	0.00	0.00	kad_op_relu

- **Performance Counter:**

	Number of Images	Time of Execution (seconds)	Clock Cycles
MLP	**100**	**25.34**	**1267396400**
CNN	**10**	**331.994**	**16599699760**

Initial Thought Process:

After analysing the profiling results of both mlp and CNN, the floating point addition and multiplication operations were identified as the functions which if implemented on the hardware would give us major speedup needed.

After the successful implementation of floating point addition and multiplication, it can be used as combinational custom instruction and reuse the same hardware in the software. These units also served as base for sdot product multicycle custom instruction and accelerator.

Since the data needed to be written on the accelerator avalon interface was large, and involved frequent memory access by the processor. This could stall other important tasks performed by the NIOS IIf processor. Hence techniques such as pipelining the accelerator, utilizing Direct Memory Access Controller for burst transfer of the data has to be explored to reduced memory accesses of the processor and to achieve the best result.

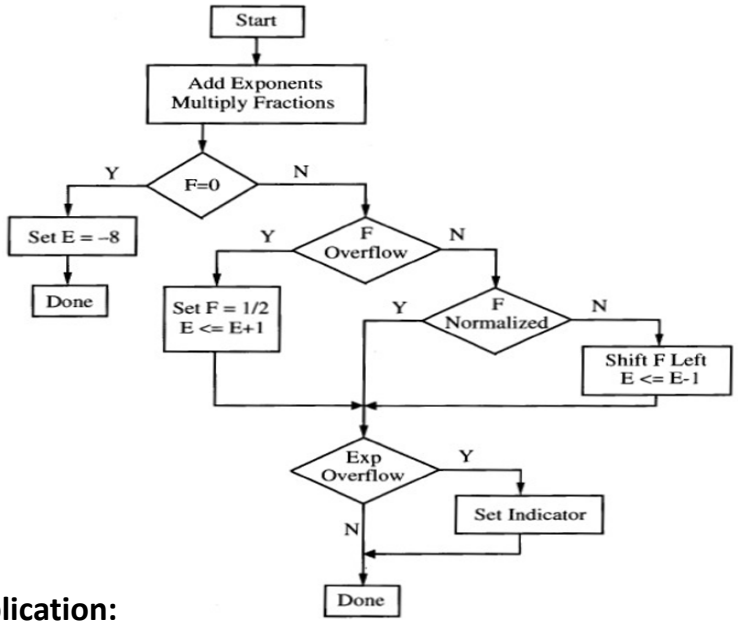
Functions Identified for Speedup:
kad_sdot, kad_saxpy_inlined, kad_sgemm_simple

Floating Point Addition and Multiplication Combinational Circuit (Basic blocks):

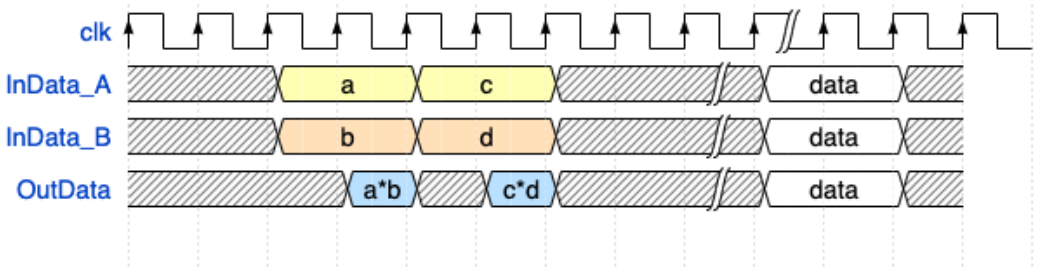
The Addition and Multiplication operations have been implemented using the *Data Flow modeling* in Verilog, both operations executed within a single clock cycle as combinational custom instruction.

The implementation ensures precision and proper care is taken to account for the corner cases, which is one of the main reasons for not using existing LPM library functions for the floating-point operations.

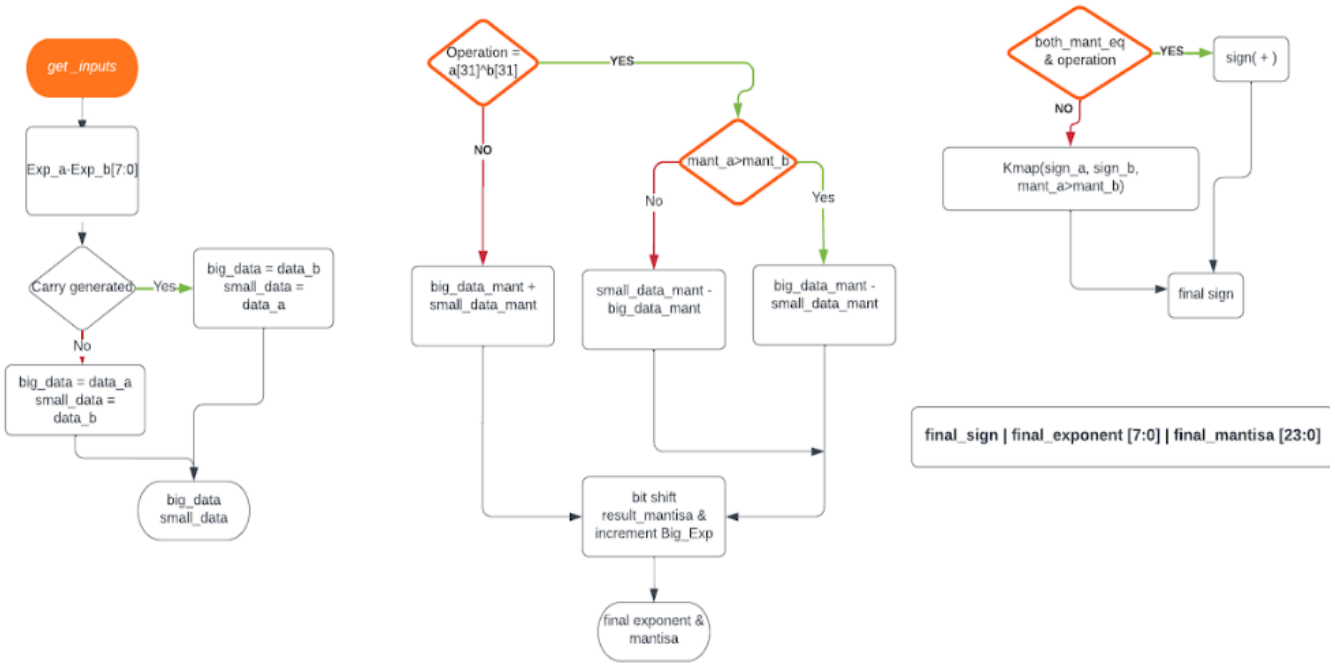
- **<u>Floating Point Multiplication data flow:</u>**



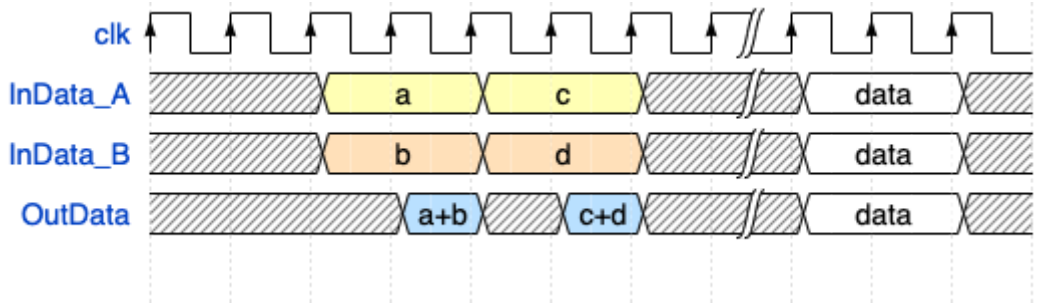
Timing Diagram for Multiplication:



- **<u>Floating Point Addition & Subtraction data flow:</u>**



Timing Diagram for Addition:

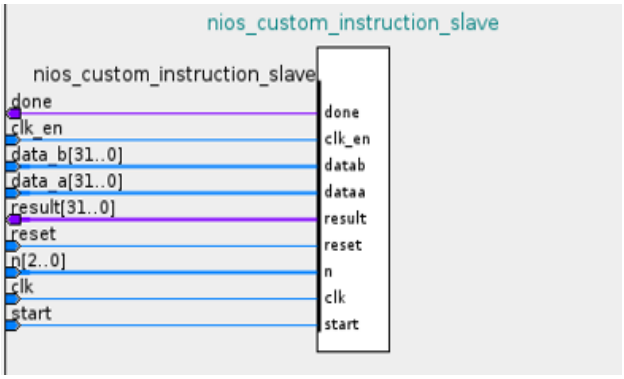


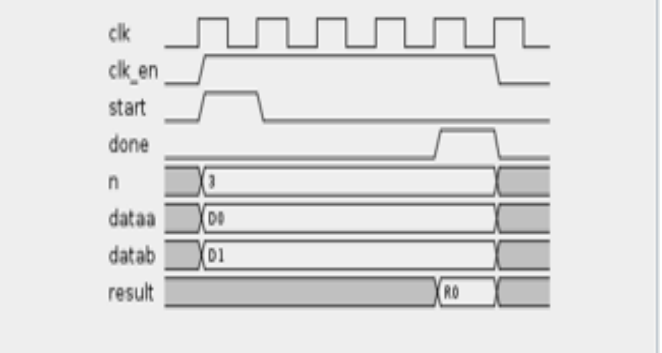
Multi-Cycle Custom Instruction for Sdot:

The single-cycle Custom Instruction modules for floating-point addition and floating-point multiplication were instantiated in the multicycle CI code with appropriate wait/delay times.

The software sends, reads, and resets the custom instruction hardware using the value of n defined in the Verilog. Currently, this is being achieved in two clock cycles, namely one cycle for addition and one cycle for multiplication.

Custom Instruction Hardware **Waveform for Custom Instruction**





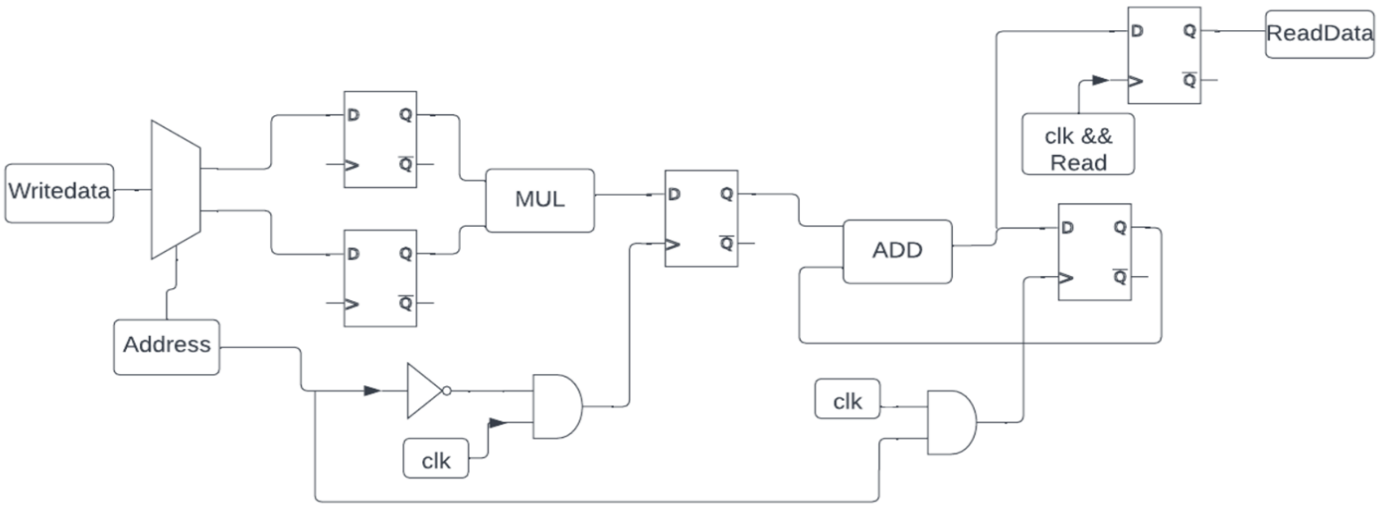
Multicycle Custom Instruction in software snippet:

```
static inline float kad_sdot(int n, const float *x, const float *y)
{
int i;
int* inputfirst  = (int*) x;
int* inputsecond = (int*) y;

for (i = 0; i < n; ++i)
      ALT_CI_SDOT_CUSTOM_INST_0(2,inputfirst[i],inputsecond[i]); //load for any n

inputfirst = &s;
*inputfirst = ALT_CI_SDOT_CUSTOM_INST_0(1,0,0);                  //read for n = 2
ALT_CI_SDOT_CUSTOM_INST_0(0,0,0);                               //reset for n = 0
}
```

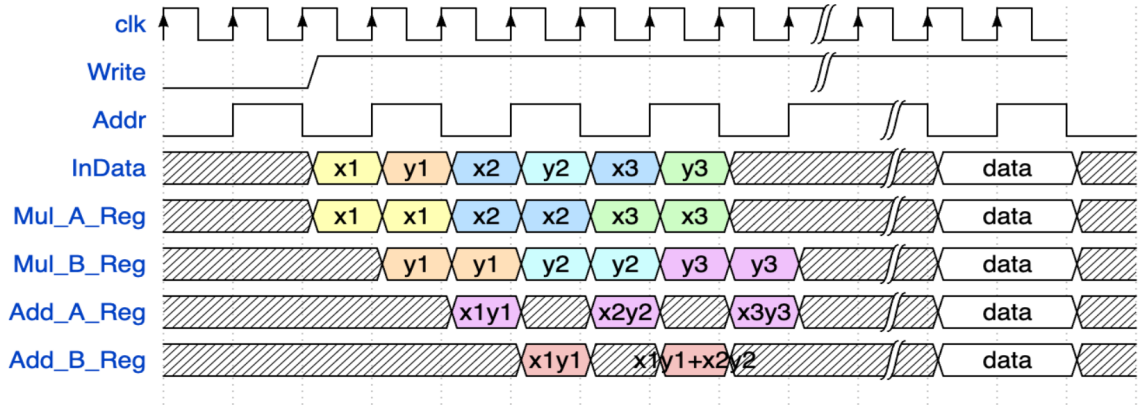
Sdot Product Accelerator pipelined

Hardware Accelerator uses pipelining to improve the throughput of dot product. It stores the inputs in two registers (MUL_A & MUL_B) based on the address specified from the processor. It leverages the parallel computation of addition and multiplication operations in two cycle.

Accelerator Block Level Implementation:



Timing Diagram:



Hardware Accelerator Software code Snippet:

```
int *regarraybase = (int *) (SDOT_ACCELERATOR_0_BASE | NIOS2_DCACHE_BYPASS_MASK);
int sdot_accel = 0;
for(int i=0;i<n;i++)
{
      *(regarraybase) = *(int*)(x+i);           //load x to accel
      *(regarraybase + 1) = *(int*)(y+i);      //load y to accel
}
*(regarraybase) = 0;                            //push the pipe
sdot_accel = *(regarraybase);                   //read the accelerator value
s = *(float*)&sdot_accel;
```

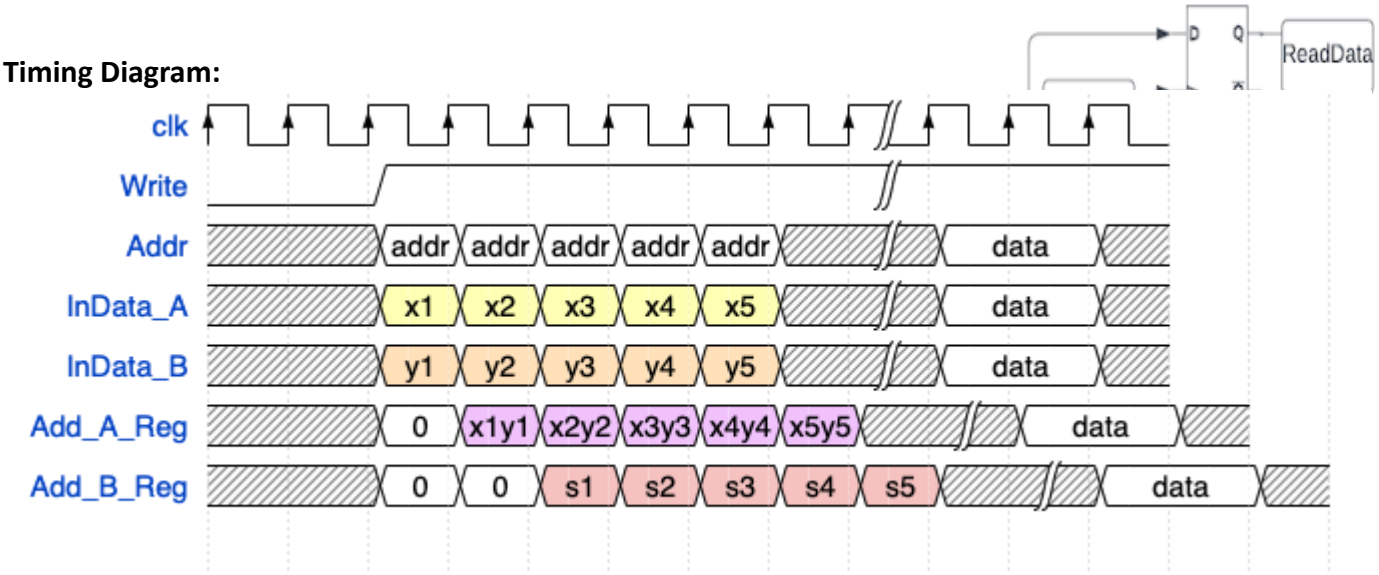
Sdot Product Accelerator with DMA transfer

Uses 256 register array bank with each register of 32 bits length, the last register is the flag register to compute or store the incoming data, pipelining is achieved by performing addition of previous input while multiplication is being performed for the recent input.

if reg[255] == 32'd0 (flag) : Store the incoming data to its respective address
else if reg[255] == 32'd1 (flag) : Compute dot product with respect to the previous stored data

Accelerator with DMA block diagram:

Timing Diagram:



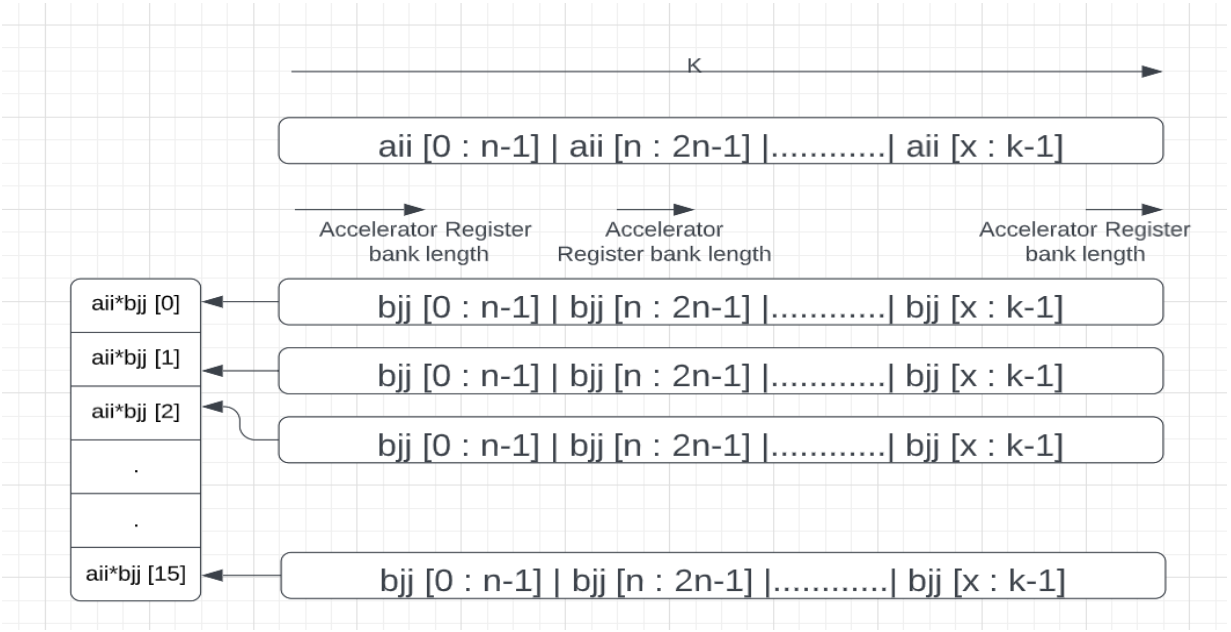
Since length of the vector x (K) won't be same as accelerator register bank lenght (n), data is divided for length n and dma transfer is done (k/n+1) times for each vector.

Optimising the SW for good use of HW (By Reducing the number of DMA transfers):

By analyzing the "kad_sdot" parent function "kad_sgemm_simple", we found dot product is performed 16 times with aii vector being constant as shown below. Hence we could exploit these known number of inner loop iteration and send the x value needed for the dot product only once to the accelerator by using DMA

```
for (ii = i; ii < ie; ++ii) { /* loop tiling */
        const float *aii = A + ii * K, *bjj;
        float *cii = C + ii * N;
        for (jj = j, bjj = B + j * K; jj < je; ++jj, bjj += K)
                cii[jj] += kad_sdot(K, aii, bjj);
}
```

Graphical representation of the solution:



K (Vector length); N (Accelerator Register Bank length);

Code snippet:

```
int final_cii =0;
int sdot_dma_temp = 0;
int cnt = 0;

int i = 0;
int max_length = K;
int n_tmp = ACCEL_REG;

while(max_length > ACCEL_REG )
{
        cnt=0;

        *(regarraybase + 255) = 0;                  //set the store flag in accelerator flag
        alt_dcache_flush(aii + ACCEL_REG*i, n_tmp*4);    //flush the aii cache
```

```
        load_dma(n_tmp, aii + ACCEL_REG*i);                //load the aii to accelerator

        *(regarraybase + 255) = 1;                    //set the compute flag in accelerator

      for(jj = j, bjj = B + j * K; jj < je; ++jj, bjj += K)
        {
                alt_dcache_flush(bjj + ACCEL_REG*i,n_tmp*4);     //flush the bii cache
                load_dma(n_tmp, bjj + ACCEL_REG*i);                 //load the bii to accelerator

                *(regarraybase) = 0;                                //push the pipeline
                sdot_dma_temp = *(regarraybase);                    //get the dotproduct(accel o/p)

                prev_val = *(sdot_int_array + cnt);
                *(sdot_int_array + cnt) = ALT_CI_FLOATING_ADD_CI_0(sdot_dma_temp, prev_val); //update 16 element
                cnt=cnt+1;
        }

        max_length = max_length - ACCEL_REG;
        n_tmp = (max_length<255)?max_length:255;
        i = i+1;
}

int  tmp = (max_length < ACCEL_REG)? max_length: n_tmp;
cnt = 0;

alt_dcache_flush(aii + ACCEL_REG*i,tmp*4);      //flush the aii cache
*(regarraybase + 255) = 0;                          //set the store flag in accelerator flag
load_dma(tmp, aii + ACCEL_REG*i);           //load the aii to accelerator

*(regarraybase + 255) = 1;                  //set the compute flag in accelerator

for(jj = j, bjj = B + j * K; jj < je; ++jj, bjj += K)
{
        alt_dcache_flush(bjj + ACCEL_REG*i,n_tmp*4);  //flush the bii cache
        load_dma(n_tmp, bjj + ACCEL_REG*i);           //load the bii to accelerator

        *(regarraybase) = 0;                          //push the pipeline
        sdot_dma_temp = *(regarraybase);           //get the dot product(accelerator) output

        prev_val = ALT_CI_FLOATING_ADD_CI_0(*(int*)(cii+jj), sdot_dma_temp); //update 16 element integer array

        sdot_dma_temp = *(sdot_int_array+cnt);
        final_cii =  ALT_CI_FLOATING_ADD_CI_0( sdot_dma_temp, prev_val);     // sum it to final cii
        *(cii+jj) = *(float*)&final_cii;
      cnt=cnt+1;
}
```

Reusing HW components for other functionality:

Hardware components such as dotproduct accelerator and combinational custom instruction are reused in the software in functions such as kad_saxpy_inlined and kad_sgemm_simple functions for the better performance.

Configuring and Using the different components:

```
#define ACCEL_REG 255 //Accelerator internal max storage

#define KAD_SAXPY_INLINED 1
//0 - SW
//1 - Custom Instruction (Addition and Multiplication separate Custom Instruction)
```

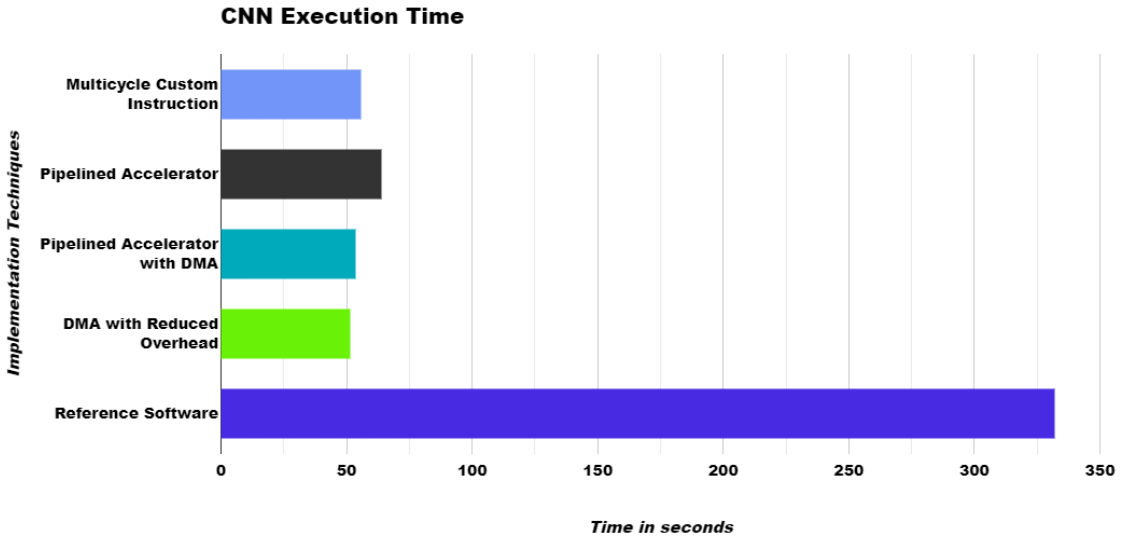
```
//2 - Accelerator (MUL_AND_ADD_ACCEL not integrated to platform designer (removed to
save area))

#define DMA_USAGE 1
// 1 - when using DMA
// 0 - Don't use DMA

#define SDOT_ACCELERATOR 3 //(K_SEGMM == 0 for this settings to work) DONT FORGET
// 0 - SW
// 1 - Multi Cycle Custom Instruction
// 2 - only accelerator
// 3 - accelerate with DMA (DMA_USAGE == 1)!!!! DONT FORGET

#define K_SEGMM 1
//0 - SW
//1 - USE DMA to load weight once to accel and calculate sdot accordingly (will
reduce 16 times overhead of sending X data in sdot product of X * Y)
```

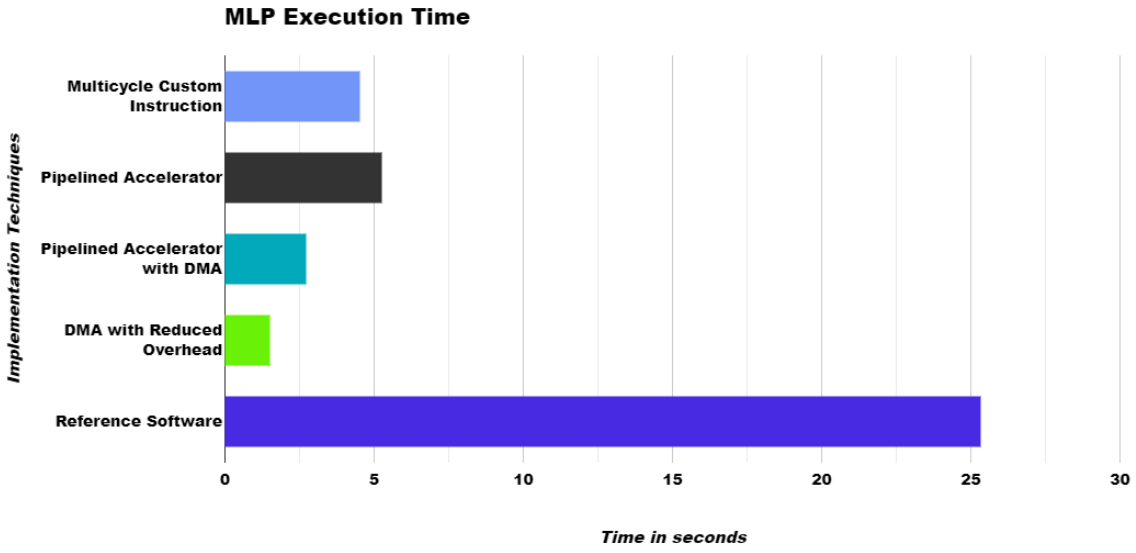
If the accelerator internal register bank can be increased then by increasing the macro "`ACCEL_REG`" according to accelerator internal memory will speedup the performance.

Benchmarking:

		MLP (100 images)	CNN (10 images)
Reference Software	**Time**	25.34	331.994
	Cycles	1267396400	16599699760
Multi Cycle Custom Instruction	**Time**	4.5346	55.8938
	Cycles	226717990	2794694522
	Speed Up	**~5.5x**	**~5.2x**
Pipelined Accelerator	**Time**	5.24693	63.9611
	Cycles	262346710	3198055129
	Speed Up	**~4.83x**	**~5.1x**
Pipelined Accelerator + DMA	**Time**	2.70741	53.7076
	Cycles	135370384	268538232
	Speed Up	**~ 9.362x**	**~ 6.18x**
DMA with reduced overhead	**Time**	1.50296	51.25944 (512.66 - 100 img)
	Cycles	75147816	2562971843
	Speed Up	**~16.86x**	**~ 6.47x**

Graphical Overview:





Analysis of the result:

We have achieved the best case speedup of 16.86x for the MLP network and 6.47x for the CNN network using pipelined accelerator and DMA implementation with reduced overhead, wherein we send the weights first in one go and exploit the case where values to be sent are constant over multiple iterations of the for loop.

NOTE:

Profiling and comparinsion for the speedup of CNN for the 100 images was not doe because, running the reference software for 100 images took long time. When **CNN** was tested with 100 images with our best implementation it took **512.66 seconds for 100 images**.

We understand that the Hardware accelerator implementation when combined with techniques such as Pipelining, DMA transfer and loop unrolling would outperform Multi cycle Custom Instruction. Since the Multicycle Custom Instruction unlike the Combinational Custom Instruction stalls the processor which is not the case with Hardware Accelerator Implementation, wherein both the Processor and HW Accelerator could work concurrently.

Project Summary:

1) Addition and Multiplication Single Cycle Combinational Circuit as Custom Instruction
2) Multi Cycle Custom Instruction for dot product computation
3) Pipelined Accelerator for Dot Product computation
4) Pipelined Accelerator for Dot Product computation with Direct Memory Access (DMA).
5) DMA implementation with reduced overhead — Best Case Speedup

Bottlenecks Faced During Implementation:

1) GPROF Profiling Error – Need to disable a HAL linker Flag for seamless execution of GPROF
2) NIOS JTAG – JTAG commands to reset and proper detection of the USB Blaster Channels in the Quartus Programmer
3) Floating Point Addition – Figure out the corner cases for Floating-Point Addition to ensure precise computation.
4) Cache Flushing – While implementing the DMA, need to flush the Cache so that the DMA reads correct and latest data from the memory, but this flushing needs to be performed at specific address locations only, to reduce time overhead.
5) Accelerator Address – Referencing the proper Accelerator address frm the SW.
6) Cache Bypass - Bypassing the Cache by setting the MSB bit of the accelerator address to 1 while accessing the HW accelerator from the NIOS.

Scope of further improvement :

Further scope of improvement could involve using a synchronous FIFO or ram array in the accelerator instead of register buffer array as an efficient Data Structure for storage along with DMA. This would entail faster data placement and even faster retrievals.
Another technique we could explore was loop unrolling in the C code implementation. Varying loop unrolling length could give us varying degrees of speedups at the cost of area overhead.
Better grip on the reference software and more importantly the API calls and code flow of the Inference process would also help us in moving towards multiple byte sized computations in parallel.