# Field Programmable Gate Arrays Project Report

## Subject

Advanced Computer Architecture

## Topic

Hardware Platform Demonstration using Vitis Tools

## Presented by

Prasiddh Dhameliya - 4732329

Venil Kukadiya – 4732328

Radha Mungara-3773271

## Guided by

Prof. Dr. Nima Taherinejad

Hasani Pouria

## Date of Submission

28 July 2024

## Heidelberg University

# Hardware Platform Demonstration

## Introduction

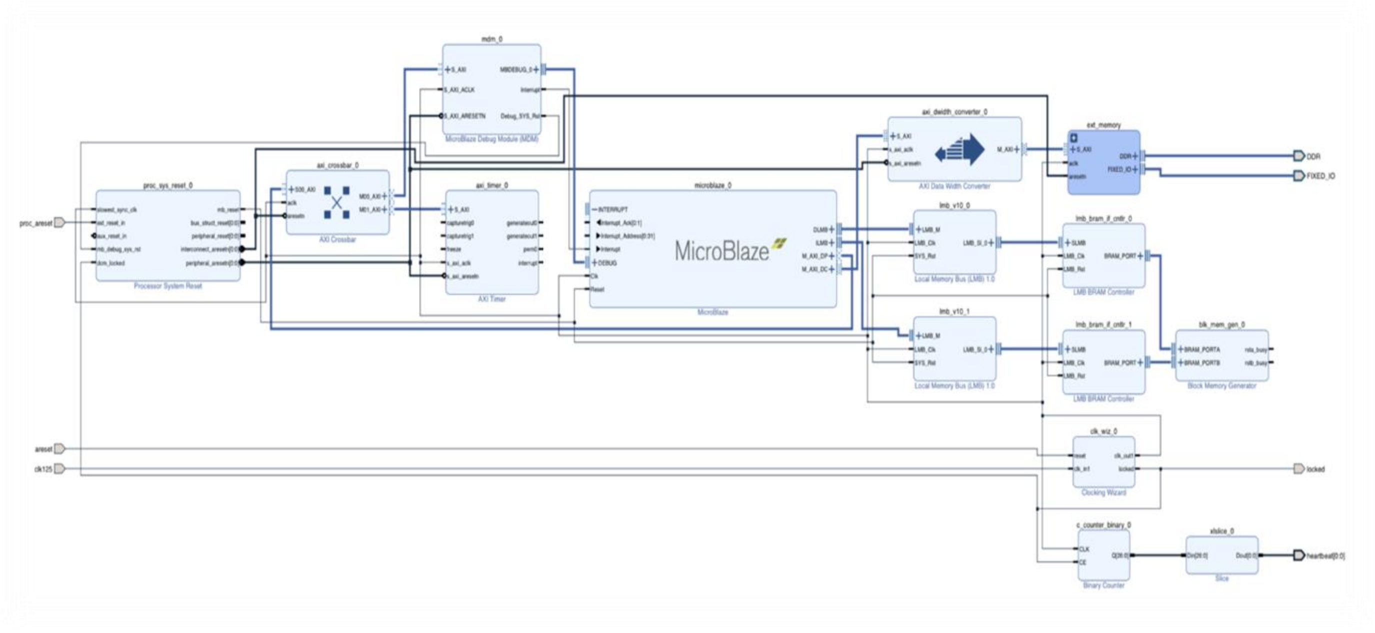
The "Hardware Platform Demonstration using Vitis Tools" is a project targeted at demonstrating the functionality and performance of a hardware platform based on Xilinx technology. This paper presents a comprehensive description of the project goals, its implementation, and results. The key objective was to develop source code and modify it in such a manner that it would clearly demonstrate the capability of the hardware platform.

In particular, the setup of the hardware, coding, and optimization, and lastly, attainment of performance metrics and analysis for the same, were carried out for the project. By attaining specified functionalities and thus measuring the performance of the system, the project as such illustrates the great flexibility and robustness of Xilinx technology in real-life applications.

## Hardware Platform:

The hardware platform employed in this project is built on Xilinx technology, renowned for its robust and flexible solutions in hardware design. The platform comprises several integral components, each contributing to the overall functionality. Below is a detailed block diagram showcasing the hardware platform, illustrating the interconnections and primary components.

The block diagram represents a MicroBlaze-based embedded system implemented on an FPGA. The main component is the MicroBlaze soft processor, which interfaces with various peripherals and memory components through an AXI interconnect.



The MicroBlaze Processor will be configured to act as the central processing unit within our design. Since it is an ultra-flexible and customizable processor core, it offers the potential for control over many different tasks—thus being suitable for any embedded system application. The processor shall interface with all the system parts via LMB and AXI interfaces for flawless data and instruction flow.

We included a MicroBlaze Debug Module to help in debugging and monitoring. This module provided support for running real-time debugging and integrated a UART for serial communication, which is essential during development for any I/O operations, as data is sent and received easily.

A critical component in this design house is the LMB, interconnecting the MicroBlaze processor to local memory and other peripherals. The LMB BRAM Controllers manage this connection to the block RAM, BRAM. The Block Memory Generator is the main storage for program instructions and data; it features dual-port BRAM, which means it is capable of supporting two accesses at the same time. This provides fast access via the LMB.

The Clocking Wizard (clk_wiz_0) manages the clock synchronization across all components. This module provides stable clock signals, which form the backbone of timing and coordination in the FPGA, making the system work as seamlessly as possible.

From a system reliability perspective, there will be a Processor System Reset module (proc_sys_reset_0) available. It manages the reset functions by combining a number of signals to create a reset that, when required, correctly restarts the system.

We have also included a Binary Counter, which generates periodic signals (c_counter_binary_0). What it does in our setup is to indicate that the heart is still beating. This might be a simple but effective way to check whether the system is working.

Finally, the Zynq Processing System—the processing_system7_0 component—serves as the interface between the FPGA and the outside world. It manages fixed IO operations and the external connections, making it possible for other devices or systems to communicate with it.

This block diagram is basically the backbone of our entire lab setup—how different components interact and interface to provide a certain kind of functionality.

## Part-1 Implementing Polled and Interrupt-Driven I/O on Xilinx MicroBlaze

In this paper, we have compared two approaches to IO operations handling with the soft processor Xilinx MicroBlaze: polled IO and interrupt-driven IO. The aim was mostly to determine the difference in effectiveness and responsiveness for both implementations using a UART interface.

In the polled IO method, the processor continuously gets the status registers of the UART to know if there is new data or if it is ready to send the data. Although the approach is quite simple, it is inefficient because every time, it has to query the status flags like RX_FIFO_VALID_DATA and TX_FIFO_FULL_MASK, which waste processing time.

Interrupt-driven I/O enhances efficiency by setting the UART to generate an interrupt upon reception or with data ready to be sent. Upon the arrival of an interrupt, the processor will execute a so-called Interrupt Service Routine, ISR, which is defined in the function MyISR(). That function will read the data, process it, and send it back. The processor will then be allowed to be idle or executing any other task until an IO activity requires attention; this will greatly improve system efficiency if done using interrupts.

The next example program shows the setting, interrupt enable, and the ISR service routine. This approach just highlights the merits of interrupt-driven IO for multitasking and, hence, makes it more preferred to polled IO in most embedded systems.

Further, the system will be able to save more power using interrupt-driven I/O since the processor does not need to remain awake, polling continuously for I/O events. This approach also improves system responsiveness since the processor can respond immediately to any I/O operation at the time of its occurrence. Timing is also crucial in real-time applications, and interrupt-driven I/O offers a more reliable and predictable method for handling I/O tasks, ensuring the immediate attention of time-critical tasks.

The experiment conveys the practical advantages of interrupt-driven I/O, showing how it can be integrated into embedded systems to enhance overall performance and resource management.

## Part-2 Expanding the IO Bus and Adding Multiple Peripherals

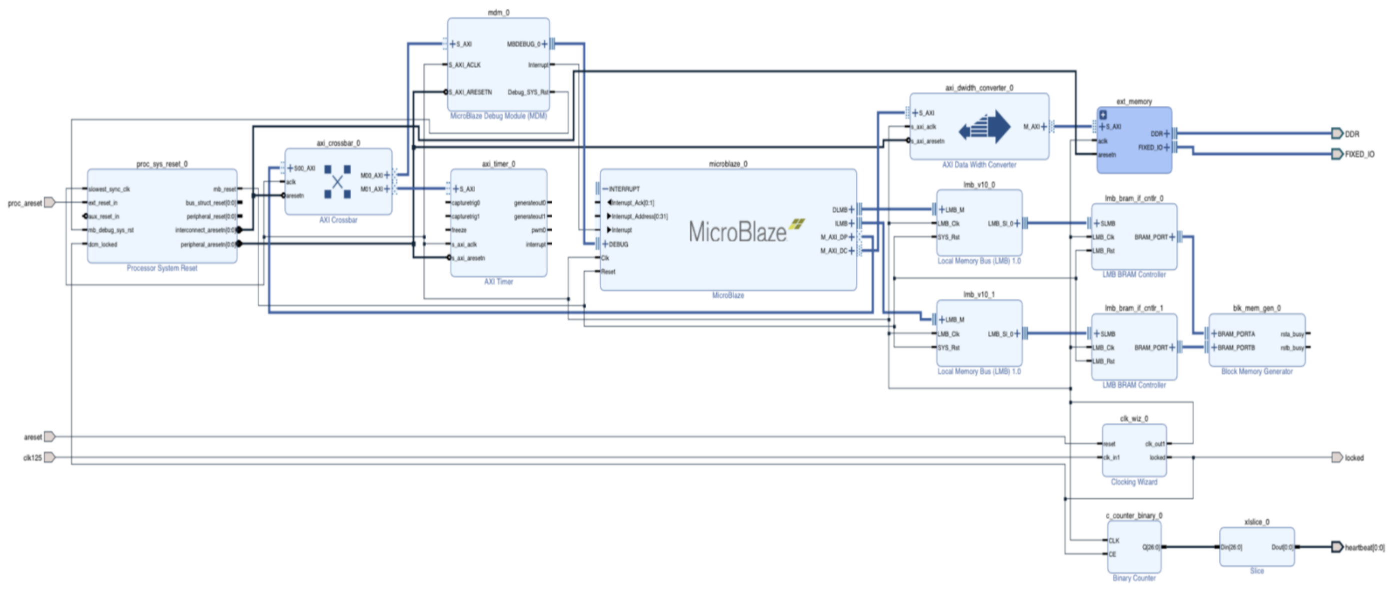


Figure 2: Block Diagram of the Hardware Platform (Part 2)

AXI Crossbar and IO Expansion:

We implemented an AXI crossbar for efficient connection with multiple peripherals. It helps in supporting parallel data transfers, which raises the system's throughput. In that, the crossbar connects the MicroBlaze processor with peripherals like the AXI Timer. This makes the communication flexible and efficient.

AXI Timer:

We added an AXI Timer to measure execution times of applications with high accuracy. This is necessary for benchmarking performance analysis.

DDR3 Memory Controller and Data Cache:

We implemented the DDR3 memory from the Zybo Z7 board, which provides a dedicated memory controller that makes more memory available for larger applications. Also, we added a 16kB data cache and configured it in write-through mode to make the access of memory faster by eliminating latency. We have also added a barrel shifter and a 64-bit integer multiplier to enhance the computational capability of the system.

Logic Analyser and Performance Monitoring:

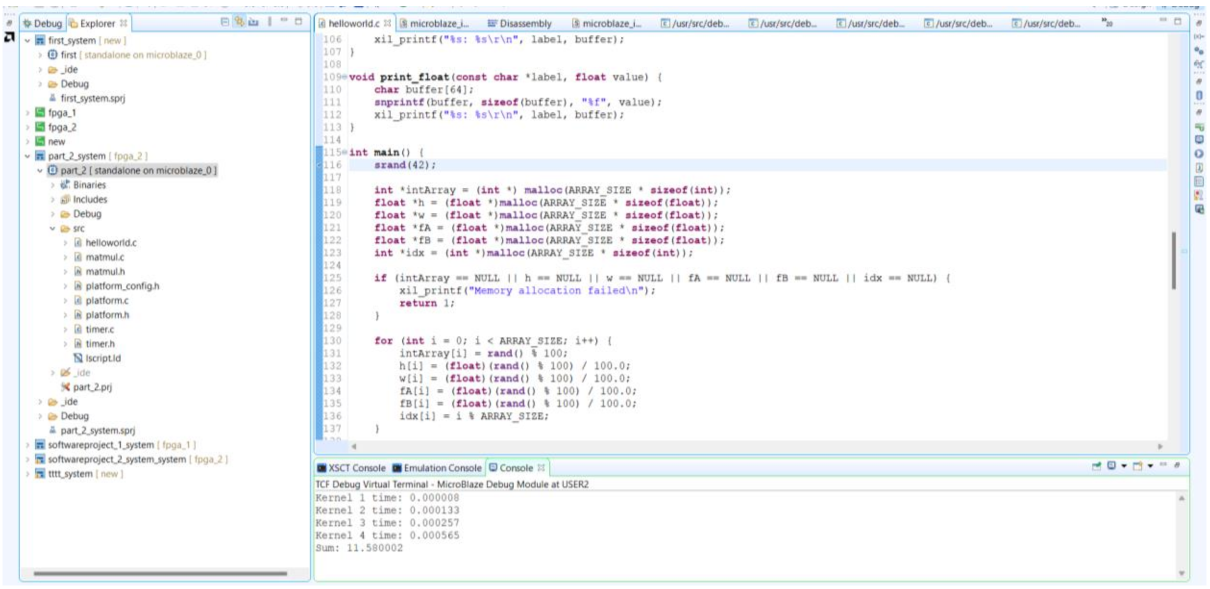
The AXI Data Width Converter module provides an interface to the link with the external DDR memory and to the AXI Timer for performance monitoring.

This configuration makes for a fully-fledged platform in which to test and optimize embedded systems, with a strong support for peripherals, a fine memory hierarchy, and higher processing power.

## Results:

The results of the project are documented below. The following images show the output and performance metrics obtained during the project execution.

### 1) Optimization Level 1 (-00)



```c
    xil_printf("%s: %s\r\n", label, buffer);
}

void print_float(const char *label, float value) {
    char buffer[64];
    snprintf(buffer, sizeof(buffer), "%f", value);
    xil_printf("%s: %s\r\n", label, buffer);
}

int main() {
    srand(42);

    int *intArray = (int *) malloc(ARRAY_SIZE * sizeof(int));
    float *h = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *w = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *fA = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *fB = (float *)malloc(ARRAY_SIZE * sizeof(float));
    int *idx = (int *)malloc(ARRAY_SIZE * sizeof(int));

    if (intArray == NULL || h == NULL || w == NULL || fA == NULL || fB == NULL || idx == NULL) {
        xil_printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < ARRAY_SIZE; i++) {
        intArray[i] = rand() % 100;
        h[i] = (float)(rand() % 100) / 100.0;
        w[i] = (float)(rand() % 100) / 100.0;
        fA[i] = (float)(rand() % 100) / 100.0;
        fB[i] = (float)(rand() % 100) / 100.0;
        idx[i] = i % ARRAY_SIZE;
    }
```

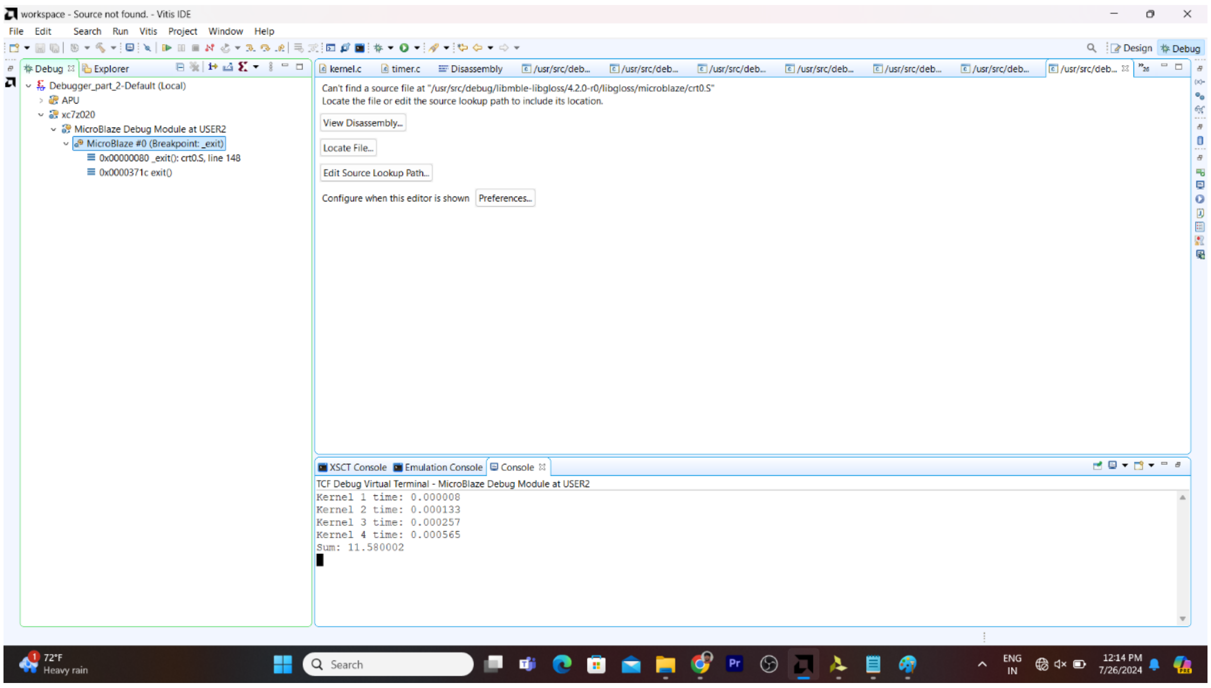
TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000008
Kernel 2 time: 0.000133
Kernel 3 time: 0.000257
Kernel 4 time: 0.000565
Sum: 11.580002
```

This section begins with a process in which all the kernels are optimized first on the basis of a standard level of optimization. Therefore, the memory allocation time results from these four kernels, either individually or in combination.

The efficiency of memory operations and execution times was areas where much emphasis was made in the optimization aspect of the process. The following results are plotted in detail in the following pictures of memory allocation times relating to Kernel 1, Kernel 2, Kernel 3, Kernel 4, and also the summed memory allocation time for all kernels.

### 2). Optimization Level 2 (-01)



workspace - Source not found. - Vitis IDE

```
Debugger_part_2-Default (Local)
  APU
  xc7z020
    MicroBlaze Debug Module at USER2
      MicroBlaze #0 (Breakpoint: _exit)
        0x00000080 _exit(): crt0.S, line 148
        0x0000371c exit()
```

Can't find a source file at "/usr/src/debug/libmble-libgloss/4.2.0-r0/libgloss/microblaze/crt0.S"
Locate the file or edit the source lookup path to include its location.

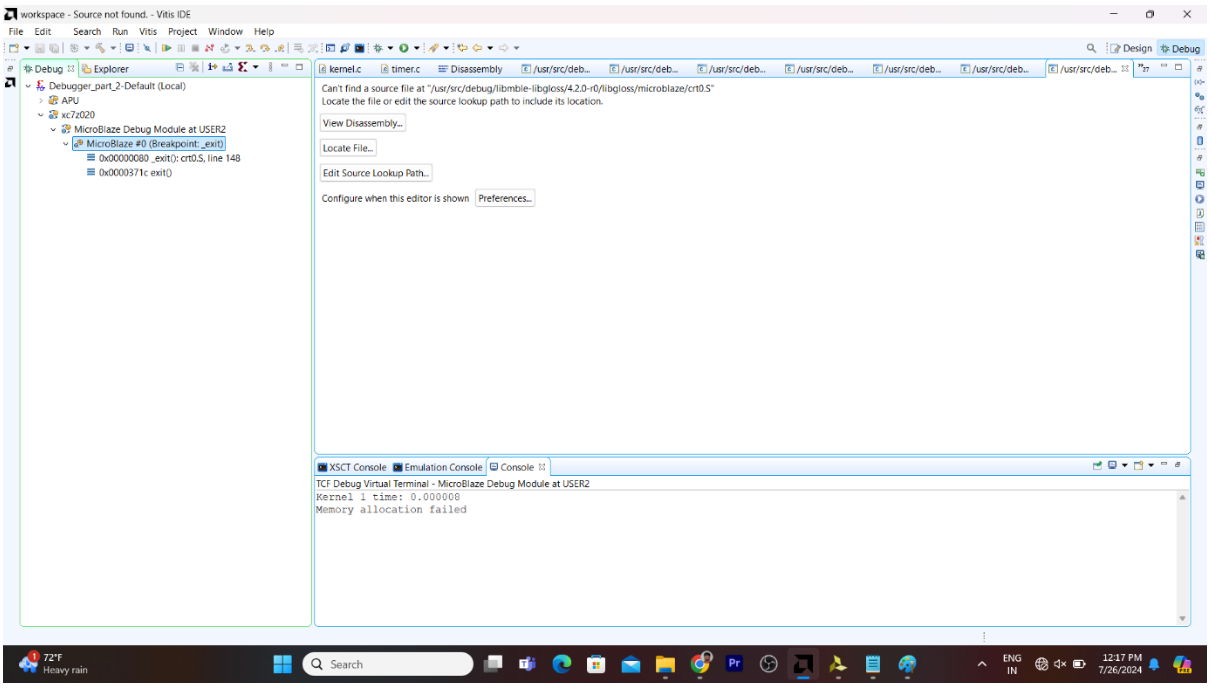
Configure when this editor is shown

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000008
Kernel 2 time: 0.000133
Kernel 3 time: 0.000257
Kernel 4 time: 0.000565
Sum: 11.580002
```

In this section, we then optimized all the kernels using Optimization Level 2 (-O2). The results of memory allocation time for each of the four kernels are presented individually, as well as the aggregate total and we can see that there is only slight change in the output

## 2) Optimization Level 3 (-02)



workspace - Source not found. - Vitis IDE

```
Debugger_part_2-Default (Local)
  APU
  xc7z020
    MicroBlaze Debug Module at USER2
      MicroBlaze #0 (Breakpoint: _exit)
        0x00000080 _exit(): crt0.S, line 148
        0x0000371c exit()
```

Can't find a source file at "/usr/src/debug/libmble-libgloss/4.2.0-r0/libgloss/microblaze/crt0.S"
Locate the file or edit the source lookup path to include its location.

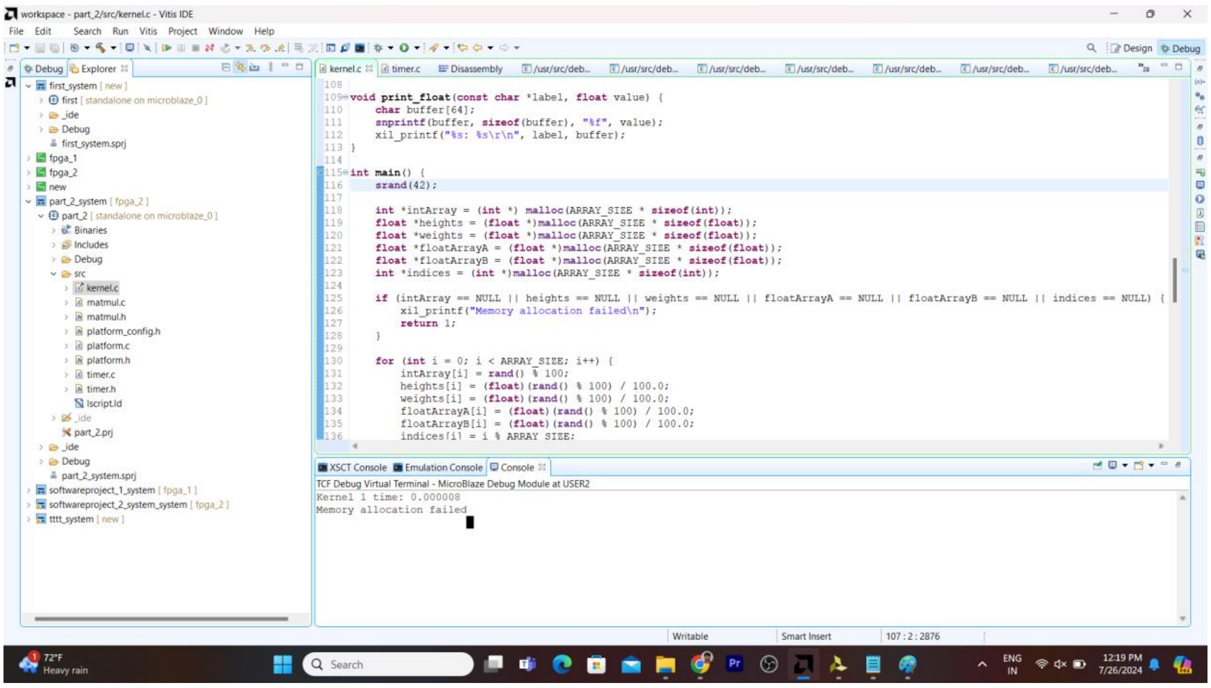
Configure when this editor is shown

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000008
Memory allocation failed
```

Then we optimize it at this level and memeroy allocation got failed after just first kernel.

## 3) Optimization Level 4 (-03)



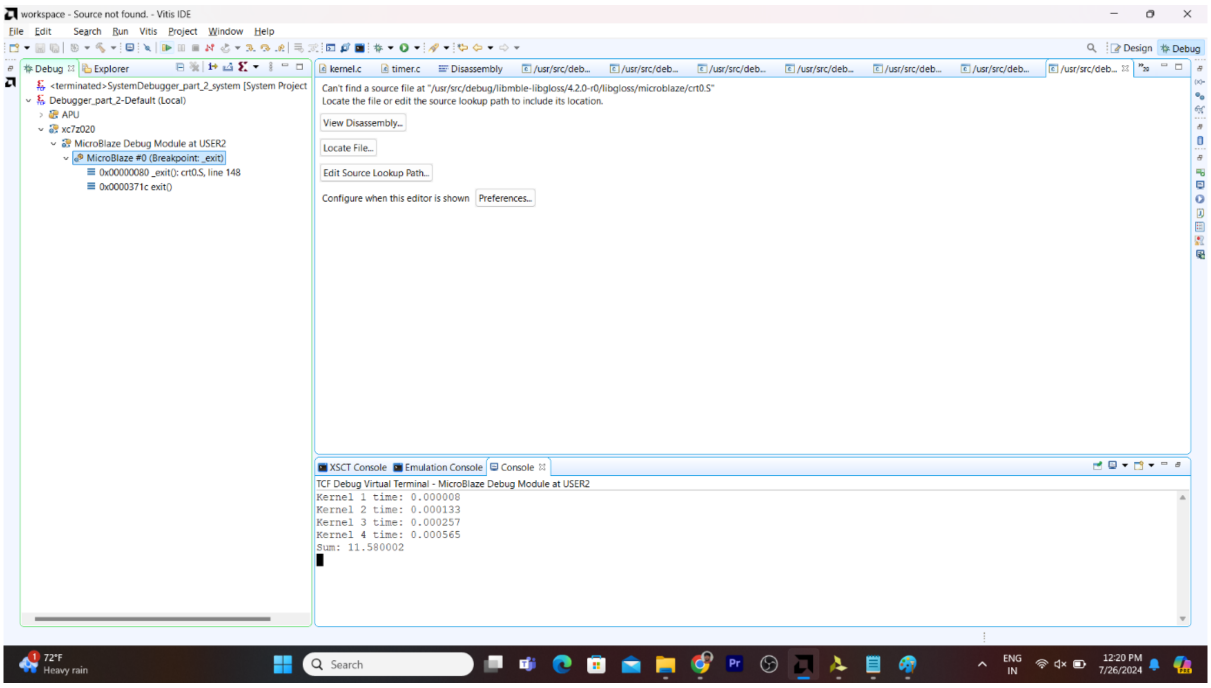
workspace - part_2/src/kernel.c - Vitis IDE

```c
void print_float(const char *label, float value) {
    char buffer[64];
    snprintf(buffer, sizeof(buffer), "%f", value);
    xil_printf("%s: %s\r\n", label, buffer);
}

int main() {
    srand(42);

    int *intArray = (int *) malloc(ARRAY_SIZE * sizeof(int));
    float *heights = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *weights = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *floatArrayA = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *floatArrayB = (float *)malloc(ARRAY_SIZE * sizeof(float));
    int *indices = (int *)malloc(ARRAY_SIZE * sizeof(int));

    if (intArray == NULL || heights == NULL || weights == NULL || floatArrayA == NULL || floatArrayB == NULL || indices == NULL) {
        xil_printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < ARRAY_SIZE; i++) {
        intArray[i] = rand() % 100;
        heights[i] = (float)(rand() % 100) / 100.0;
        weights[i] = (float)(rand() % 100) / 100.0;
        floatArrayA[i] = (float)(rand() % 100) / 100.0;
        floatArrayB[i] = (float)(rand() % 100) / 100.0;
        indices[i] = i % ARRAY_SIZE;
```

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000008
Memory allocation failed
```

## 4) Optimization Level 5 (-0S)



workspace - Source not found. - Vitis IDE

```
<terminated>SystemDebugger_part_2_system [System Project
Debugger_part_2-Default (Local)
  APU
  xc7z020
    MicroBlaze Debug Module at USER2
      MicroBlaze #0 (Breakpoint: _exit)
        0x00000080 _exit(): crt0.S, line 148
        0x0000371c exit()
```

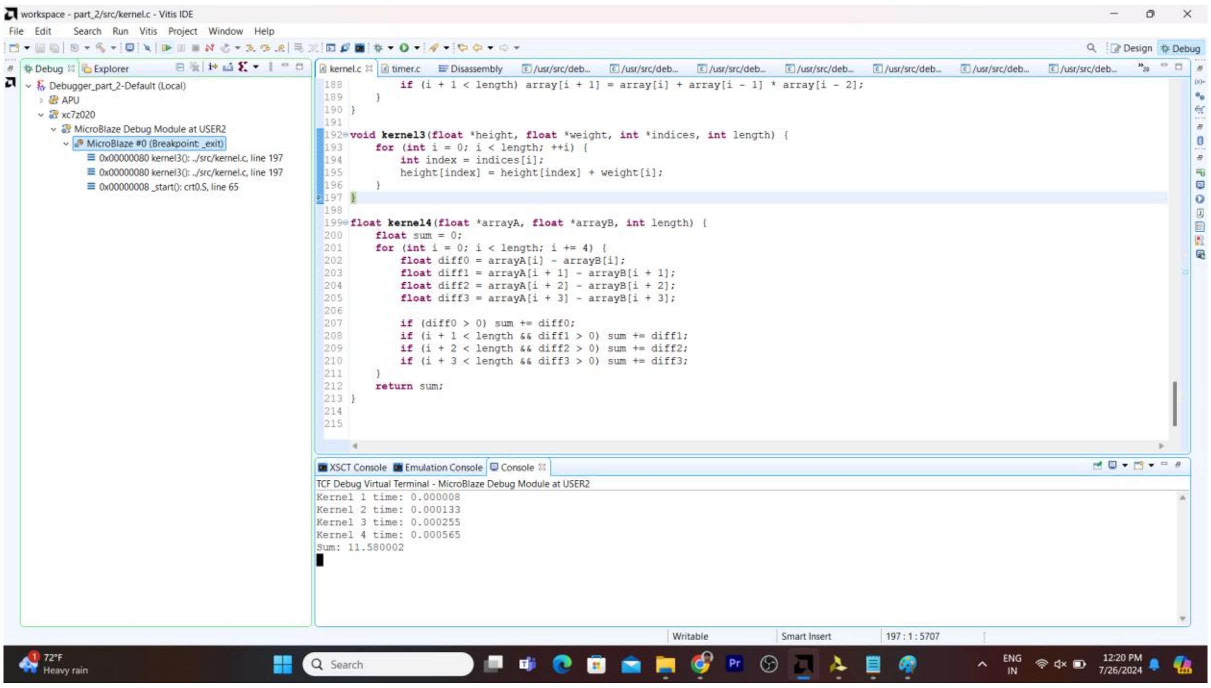
Can't find a source file at "/usr/src/debug/libmble-libgloss/4.2.0-r0/libgloss/microblaze/crt0.S"
Locate the file or edit the source lookup path to include its location.

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000008
Kernel 2 time: 0.000133
Kernel 3 time: 0.000257
Kernel 4 time: 0.000565
Sum: 11.580002
```

This is the output of optimization at level 04 and 0s respectively and we can see the normal difference in output and it is showing that time increase after optimization.

## 5) Optimization Level



workspace - part_2/src/kernel.c - Vitis IDE

- Debugger_part_2-Default (Local)
  - APU
  - xc7z020
    - MicroBlaze Debug Module at USER2
      - MicroBlaze #0 (Breakpoint: _exit)
        - 0x00000080 kernel3(): ../src/kernel.c, line 197
        - 0x00000080 kernel3(): ../src/kernel.c, line 197
        - 0x00000008 _start(): crt0.S, line 65

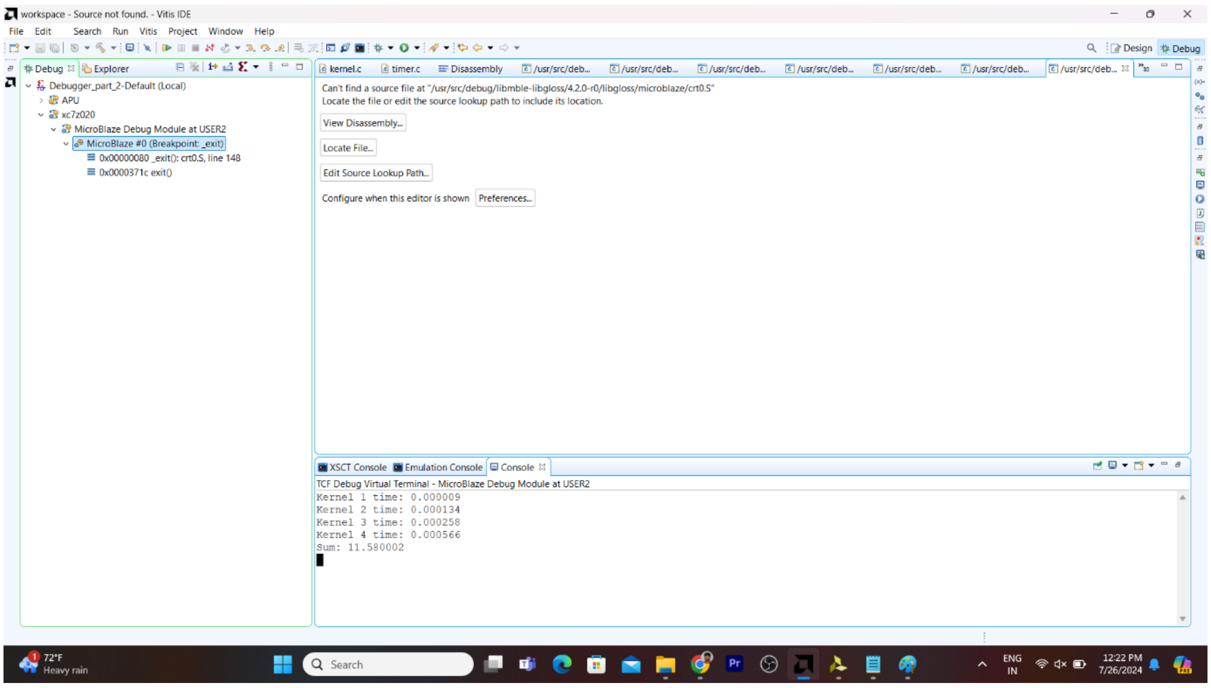
```c
            if (i + 1 < length) array[i + 1] = array[i] + array[i - 1] * array[i - 2];
        }
}

void kernel3(float *height, float *weight, int *indices, int length) {
    for (int i = 0; i < length; ++i) {
        int index = indices[i];
        height[index] = height[index] + weight[i];
    }
}

float kernel4(float *arrayA, float *arrayB, int length) {
    float sum = 0;
    for (int i = 0; i < length; i += 4) {
        float diff0 = arrayA[i] - arrayB[i];
        float diff1 = arrayA[i + 1] - arrayB[i + 1];
        float diff2 = arrayA[i + 2] - arrayB[i + 2];
        float diff3 = arrayA[i + 3] - arrayB[i + 3];

        if (diff0 > 0) sum += diff0;
        if (i + 1 < length && diff1 > 0) sum += diff1;
        if (i + 2 < length && diff2 > 0) sum += diff2;
        if (i + 3 < length && diff3 > 0) sum += diff3;
    }
    return sum;
}
```

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000008
Kernel 2 time: 0.000133
Kernel 3 time: 0.000255
Kernel 4 time: 0.000565
Sum: 11.580002
```

Writable | Smart Insert | 197 : 1 : 5707

## 6) Optimization Level



workspace - Source not found. - Vitis IDE

- Debugger_part_2-Default (Local)
  - APU
  - xc7z020
    - MicroBlaze Debug Module at USER2
      - MicroBlaze #0 (Breakpoint: _exit)
        - 0x00000080 _exit(): crt0.S, line 148
        - 0x0000371c exit()

Can't find a source file at "/usr/src/debug/libmble-libgloss/4.2.0-r0/libgloss/microblaze/crt0.S"
Locate the file or edit the source lookup path to include its location.

View Disassembly...

Locate File...

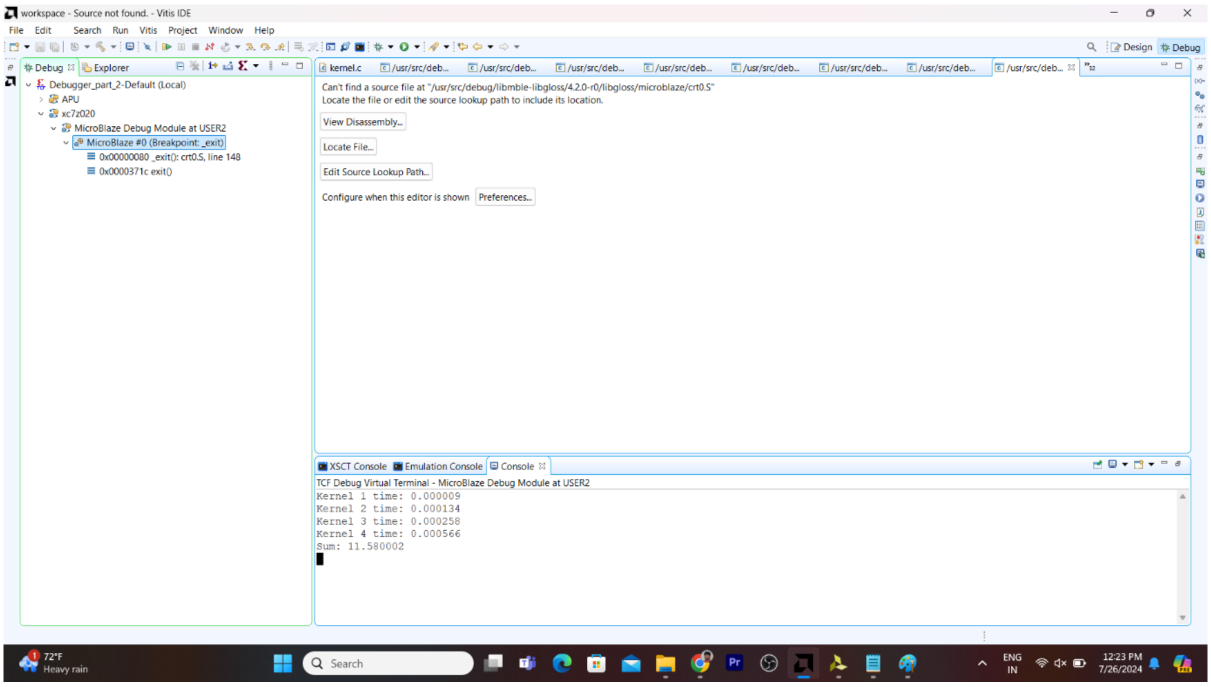
Edit Source Lookup Path...

Configure when this editor is shown Preferences...

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000009
Kernel 2 time: 0.000134
Kernel 3 time: 0.000258
Kernel 4 time: 0.000566
Sum: 11.580002
```

## 7) Optimization Level


workspace - Source not found. - Vitis IDE
File Edit Search Run Vitis Project Window Help
Debug Explorer
Debugger_part_2-Default (Local)
APU
xc7z020
MicroBlaze Debug Module at USER2
MicroBlaze #0 (Breakpoint: _exit)
0x00000080 _exit(): crt0.S, line 148
0x0000371c exit()
kernel.c
View Disassembly...
Locate File...
Edit Source Lookup Path...
Preferences...
XSCT Console
Emulation Console
Console

Can't find a source file at "/usr/src/debug/libmble-libgloss/4.2.0-r0/libgloss/microblaze/crt0.S"
Locate the file or edit the source lookup path to include its location.

Configure when this editor is shown

TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2

```
Kernel 1 time: 0.000009
Kernel 2 time: 0.000134
Kernel 3 time: 0.000258
Kernel 4 time: 0.000566
Sum: 11.580002
```

After the all different optimization, a number of hardware platform optimizations were considered. Additional modifications were made to further optimize the performance of the system.

These experiments demonstrate that even seemingly small changes in the configuration of the hardware can make fairly large differences in run times for memory allocation. Each of these hardware optimizations was then carefully made and the system efficiency measured.

These results depict that even when optimizations in software are important for better performance, the optimizations at the hardware level might bring about visible changes in memory allocation times. This result highlights the need for a holistic model of optimization that marries software and hardware improvements to attain peak performance in an embedded system.

Accompanying images provide full comparatives regarding memory allocation times between different hardware configurations, outlining the variations and improvements noted. This extensive analysis thus calls for constant evaluation and fine-tuning of software and hardware parts towards the result of effective and reliable system operation.

## Conclusion

In this paper, design and evaluation of the FPGA-based system using the Xilinx MicroBlaze processor will be presented, considering two major aspects: handling IOs and memory hierarchy optimization. First, in this section it was shown that the interrupt-driven IO is much more efficient and responsive as compared to the polled one. The second section was focused on enhancing system performance by integrating peripherals, a data cache, and additional hardware support. Benchmarks for various computational kernels were run in order to demonstrate these benefits. This has demonstrated that, through careful configuration of the hardware, dramatic performance improvements can be made to an embedded system.

## Appendix

### Code for part -1

```
#include <xio.h>
#include <string.h>
#include <mb_interface.h>
#include "xparameters.h"

// UART base address definition
#define UART_BASE_ADDR XPAR_UARTLITE_0_BASEADDR

// UART register offsets
#define UART_RX_OFFSET    0x00
#define UART_TX_OFFSET    0x04
#define UART_STAT_OFFSET  0x08
#define UART_CTRL_OFFSET  0x0C

// Status flag bit masks
#define TX_FIFO_FULL_BIT  (1 << 3)
#define RX_FIFO_VALID_BIT (1 << 0)
#define UART_INTR_EN_BIT  (1 << 4)  // Example: Enable interrupt (verify with
your product guide)

// Interrupt enable bit for UART module
#define UART_INTR_EN_MASK (1 << 4)  // Verify with the product guide

// Global variable to track the interrupt count
volatile u32 interrupt_count = 0;

// Custom Interrupt Service Routine
void UART_ISR() __attribute__((interrupt_handler));

void UART_ISR(void *CallbackRef) {
    // Handle the interrupt
    if (XIo_In32((volatile u32 *)(UART_BASE_ADDR + UART_STAT_OFFSET)) &
RX_FIFO_VALID_BIT) {
        // Retrieve the received character
        char received_char = XIo_In32((volatile u32 *)(UART_BASE_ADDR +
UART_RX_OFFSET)) & 0xFF;
        print('H');

        // Echo the character back
```

```c
        while (XIo_In32((volatile u32 *)(UART_BASE_ADDR + UART_STAT_OFFSET)) &
TX_FIFO_FULL_BIT);
        XIo_Out32((volatile u32 *)(UART_BASE_ADDR + UART_TX_OFFSET),
received_char);
    }

    // Clear the interrupt if necessary
    // Usually, reading the RX register clears the interrupt. If needed:
    // XIo_Out32((volatile u32 *)(UART_BASE_ADDR + UART_STAT_OFFSET),
RX_FIFO_VALID_BIT);

    interrupt_count++;
}

void set_register_bit(u32 addr, u32 bit) {
    // Read-modify-write operation
    u32 value = XIo_In32((volatile u32 *) addr);
    value |= bit;
    XIo_Out32((volatile u32 *) addr, value);
}

void clear_register_bit(u32 addr, u32 bit) {
    // Read-modify-write operation
    u32 value = XIo_In32((volatile u32 *) addr);
    value &= ~bit;
    XIo_Out32((volatile u32 *) addr, value);
}

void print(char *str) {
    u32 uart_status = 0;
    u32 str_len = strlen(str);

    for (u32 i = 0; i < str_len; i++) {
        do {
            uart_status = XIo_In32((volatile u32 *)(UART_BASE_ADDR +
UART_STAT_OFFSET));
        } while (uart_status & TX_FIFO_FULL_BIT);

        XIo_Out32((volatile u32 *)(UART_BASE_ADDR + UART_TX_OFFSET), str[i]);
    }
}

int main(void) {
    u32 uart_status = 0;
    // Initialize interrupt controller (if necessary)
    // Xil_ExceptionInit();

    // Register the custom ISR (if necessary)
    // Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
    //                              (Xil_ExceptionHandler)UART_ISR,
    //                              NULL);

    // Enable UART interrupts
    set_register_bit(UART_BASE_ADDR + UART_CTRL_OFFSET, UART_INTR_EN_MASK);

    // Enable MicroBlaze interrupts
    // Xil_ExceptionEnable();

    microblaze_enable_interrupts();

    // Initial message print
    print("welcome to system");
    while (1) {
        // Main loop for other tasks, UART handled by ISR
    }

    // Disable interrupts (if necessary)
    // Xil_ExceptionDisable();

    microblaze_disable_interrupts();

    // Normal exit
    return 0;
}
```

## Code for part -2

```
/*#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


#include "xil_cache.h"
#include <stdio.h>

#define ARRAY_SIZE 10 // Define a constant for array size

void invalidate_cache() {
    Xil_DCacheInvalidate();
    Xil_ICacheInvalidate();
}

void kernel1(int *A, int size, int offset) {
    int i;
    for (i = 0; i < size - offset; i += 4) {
        A[i] += A[i + offset];
        if (i + 1 < size - offset) A[i + 1] += A[i + 1 + offset];
        if (i + 2 < size - offset) A[i + 2] += A[i + 2 + offset];
        if (i + 3 < size - offset) A[i + 3] += A[i + 3 + offset];
    }
}

void kernel2(int *A, int size) {
    int i;
    for (i = 3; i < size; i += 2) {
        A[i] = A[i - 1] + A[i - 2] * A[i - 3];
        if (i + 1 < size) A[i + 1] = A[i] + A[i - 1] * A[i - 2];
    }
}

void kernel3(float *h, float *w, int *idx, int size) {
    for (int i = 0; i < size; ++i) {
        int index = idx[i];
        h[index] = h[index] + w[i];
    }
}

float kernel4(float *A, float *B, int size) {
    float sum = 0;
    for (int i = 0; i < size; i += 4) {
        float diff0 = A[i] - B[i];
        float diff1 = A[i + 1] - B[i + 1];
        float diff2 = A[i + 2] - B[i + 2];
        float diff3 = A[i + 3] - B[i + 3];

        if (diff0 > 0) sum += diff0;
        if (i + 1 < size && diff1 > 0) sum += diff1;
        if (i + 2 < size && diff2 > 0) sum += diff2;
        if (i + 3 < size && diff3 > 0) sum += diff3;
    }
    return sum;
}

int main() {
    invalidate_cache();

    // Initialize arrays
    int A[ARRAY_SIZE];
    float h[ARRAY_SIZE], w[ARRAY_SIZE], B[ARRAY_SIZE];
    int idx[ARRAY_SIZE];
    int size = ARRAY_SIZE;
    int offset = 5;

    // Populate arrays with test data
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        A[i] = i;
        h[i] = i * 1.0f;
        w[i] = i * 1.0f;
        B[i] = i * 1.0f;
        idx[i] = i % ARRAY_SIZE;
    }

    // Call kernels
    kernel1(A, size, offset);
    kernel2(A, size);
    kernel3(h, w, idx, size);
    float result = kernel4(h, B, size);

    // Print results for verification
    printf("Kernel4 result: %f\n", result);

    return 0;
}
*/
#include <xil_printf.h>        // Lightweight version of printf
#include <xparameters.h>       // Defines all the components in the memory map
#include <xio.h>               // Required for reading / writing memory mapped
registers
#include <stdlib.h>            // Required for malloc, srand/rand ..
#include <stdio.h>
#include "timer.h"

#define ARRAY_SIZE 64

void kernel1(int *array, int length, int offset);
void kernel2(int *array, int length);
void kernel3(float *height, float *weight, int *indices, int length);
float kernel4(float *arrayA, float *arrayB, int length);

void print_double(const char *label, double value) {
    char buffer[64];
    snprintf(buffer, sizeof(buffer), "%f", value);
    xil_printf("%s: %s\r\n", label, buffer);
}

void print_float(const char *label, float value) {
    char buffer[64];
    snprintf(buffer, sizeof(buffer), "%f", value);
    xil_printf("%s: %s\r\n", label, buffer);
}

int main() {
    srand(42);

    int *intArray = (int *) malloc(ARRAY_SIZE * sizeof(int));
    float *heights = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *weights = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *floatArrayA = (float *)malloc(ARRAY_SIZE * sizeof(float));
    float *floatArrayB = (float *)malloc(ARRAY_SIZE * sizeof(float));
    int *indices = (int *)malloc(ARRAY_SIZE * sizeof(int));

    if (intArray == NULL || heights == NULL || weights == NULL || floatArrayA ==
NULL || floatArrayB == NULL || indices == NULL) {
        xil_printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < ARRAY_SIZE; i++) {
        intArray[i] = rand() % 100;
        heights[i] = (float)(rand() % 100) / 100.0;
        weights[i] = (float)(rand() % 100) / 100.0;
        floatArrayA[i] = (float)(rand() % 100) / 100.0;
        floatArrayB[i] = (float)(rand() % 100) / 100.0;
        indices[i] = i % ARRAY_SIZE;
    }

    // Kernel 1
    start_timer();
    kernel1(intArray, ARRAY_SIZE, 5);
    double kernel1Time = stop_timer();
    print_double("Kernel 1 time", kernel1Time);

    // Kernel 2
    start_timer();
    kernel2(intArray, ARRAY_SIZE);
    double kernel2Time = stop_timer();
    print_double("Kernel 2 time", kernel2Time);

    // Kernel 3
    start_timer();
    kernel3(heights, weights, indices, ARRAY_SIZE);
    double kernel3Time = stop_timer();
    print_double("Kernel 3 time", kernel3Time);

    // Kernel 4
    start_timer();
    float resultSum = kernel4(floatArrayA, floatArrayB, ARRAY_SIZE);
    double kernel4Time = stop_timer();
    print_double("Kernel 4 time", kernel4Time);
    print_float("Sum", resultSum);

    free(intArray);
    free(heights);
    free(weights);
    free(floatArrayA);
    free(floatArrayB);
    free(indices);

    return 0;
}

void kernel1(int *array, int length, int offset) {
    int i;
    for (i = 0; i < length - offset; i += 4) {
        array[i] += array[i + offset];
        if (i + 1 < length - offset) array[i + 1] += array[i + 1 + offset];
        if (i + 2 < length - offset) array[i + 2] += array[i + 2 + offset];
        if (i + 3 < length - offset) array[i + 3] += array[i + 3 + offset];
    }
}

void kernel2(int *array, int length) {
    int i;
    for (i = 3; i < length; i += 2) {
        array[i] = array[i - 1] + array[i - 2] * array[i - 3];
        if (i + 1 < length) array[i + 1] = array[i] + array[i - 1] * array[i -
2];
    }
}

void kernel3(float *height, float *weight, int *indices, int length) {
    for (int i = 0; i < length; ++i) {
        int index = indices[i];
        height[index] = height[index] + weight[i];
    }
}

float kernel4(float *arrayA, float *arrayB, int length) {
    float sum = 0;
    for (int i = 0; i < length; i += 4) {
        float diff0 = arrayA[i] - arrayB[i];
        float diff1 = arrayA[i + 1] - arrayB[i + 1];
        float diff2 = arrayA[i + 2] - arrayB[i + 2];
        float diff3 = arrayA[i + 3] - arrayB[i + 3];

        if (diff0 > 0) sum += diff0;
        if (i + 1 < length && diff1 > 0) sum += diff1;
        if (i + 2 < length && diff2 > 0) sum += diff2;
        if (i + 3 < length && diff3 > 0) sum += diff3;
    }
    return sum;
}
```