XThread: A Hardware-based Multi-core Parallel Programming Environment Using PThread-like Programming Model

University name: Eskisehir Technical University
List of group members: Fatma Özüdoğru, Ersin Abbasoğlu
Supervisor: Assist. Prof. Dr. Ismail San
Email of students and Supervisor: ozudogrufatma26@gmail.com
ersinabbasoglu@anadolu.edu.tr isan@eskisehir.edu.tr
Which board you used: Kintex UltraScale FPGA KCU105
GitHub Link: https://github.com/DHLSan/XThread
Video Link: https://youtu.be/uplrkUejjHI

Abstract

Multi-core programming models including PThread, OpenMP and MPI allow one to utilize the power of all the existing available cores inside CPUs and GPUs. However, the number of cores, the architecture of execution units in ALUs and memory architectures are fixed in these computing platforms. Programmer needs to describe the program in a way to distribute execution tasks across available cores, with efficient data synchronization, considering the number of available cores. The achievable parallelism is limited with the fixed hardware, it may not be possible to achieve the maximum parallelism of the application. In our study, we propose XThread, a reconfigurable parallel programming framework that allows one to design and implement application-specific hardware for the application of interest from a PThread like C-level programming to overcome the limitation of the fixed hardware architectures. XThread shows a speedup 2.32x on block matrix multiplication. Considering the frequency difference between KCU105 Xilinx Kintex UltraScale FPGA and Intel i7 7th generation processor, the XThread library interface is much more advantageous in all experiments. It provides less power consumption. XThread consists of the same programming structures in PThread and easy to transfer existing PThread programs in order to speed-up application in reconfigurable hardware, e.g., FPGA.

1. Introduction

High-level synthesis (HLS) makes the FPGA programming easier by allowing the designer to describe the computation in high-level languages (C/C++) and improve the design productivity. HLS tool converts the program from untimed sequential version to cycle-accurate register-transfer level (RTL) hardware architecture. The quality of hardware design generated by HLS tools are getting better with the current research in the field. Thanks to high-level description of hardware designs, now complex hardware architectural ideas can be experimented within a relatively less time.

FPGA programming is difficult since one needs to design all the hardware modules and schedule them cycle-by-cycle at the RTL-level. RTL design offers power to a hardware designer, so that the hardware designer can control every cycle of the application. However, design productivity is normally very low when using RTL design. Indeed, designing a customized multi-FPGA architecture for each application is a very time-consuming process with low-level programming languages. Today, there are several high-level synthesis (HLS) tools that are LegUp [1] from University of Toronto, Xilinx Vivado HLS [2], Intel's HLS Compiler [3] and Bambu [4] from Politecnico di Milano. These tools enable hardware and software designers to generate register-transfer-level (RTL) hardware architectures from a function written in the C/C++ high-level language, which can reduce the design time for efficient application-specific hardware. HLS tools make FPGA programming easier and improve design productivity.

Multi-core programming models including PThread, OpenMP and MPI allow one to utilize the power of all the existing available cores inside CPUs or GPUs. Programmers need to describe the program in a way to

distribute execution tasks across available cores, with efficient data synchronization, for parallel execution in order to improve the overall performance of the application. However, the number of cores, the hardware architecture of execution units in ALUs, the instruction set architecture (ISA), and memory architectures are fixed in these computing platforms. Fixed hardware architecture limits the achievable parallelism; thus it may not be possible to achieve the maximum parallelism of the application. The aim of this project is to create a parallel reconfigurable HLS-based framework using a very similar way to PThread parallel programming environment, in order to achieve efficient, low-power, high-performance and multi-core implementations with minimum hardware knowledge to the programmer.

Many studies have been conducted on this subject and there are ongoing studies. Since engineering problems are becoming more complex, each step needs an abstraction layer. Motivation behind HLS is based on providing an abstraction between C-Level programming and RT-Level programming. Therefore, HLS is increasing production speed by letting individuals to be working on more human-like languages. On one step further, the concept of multi-thread programming and its practical studies are very important but they are not supported in the HLS environment as default. Although, there are several works that offer different approaches to multi-thread software programming for FPGAs. The work in [5] offers a framework for both PThread and OpenMP. Within their framework, a C code written by using PThread or/and OpenMP can be automatically synthesized into parallel hardware. Their framework is built on LegUp HLS. They have their own customizable HLS tool; therefore they have full access on compiler level. This makes automatic synthesize possible which is great. The work [6] is also using the LegUp HLS tool. It implemented an automated process to simplify and/or remove memory arbitrators in circuits created by this tool for multi-threaded software code that include PThread library. In our work, we do not provide an automatic process. Instead, we assume that it is incumbent on the programmer to follow our workflow. In the work [7], authors are able to generate OpenMP pragmas into Handel-C and VHDL. Their work has some strict memory limitations. They do not support global variables due to the absence of a memory system in VHDL. Our work does not have any memory limitations. We let the programmer use all accessible memory. Local variables in functions are already wrapped safely by the HLS tool and it's the responsibility of the programmer to organize global memory. Warp processing is where an on-chip processor dynamically remaps critical code regions from processor instructions to FPGA circuits at runtime [8]. The work [9] presents an additional study over prior work and proposes a new dynamic multiprocessor optimization technique which is called thread warping that uses one processor to synthesize threads into circuits on FPGAs. They use an on-chip CAD tool to create a custom hardware accelerator for a given function. Their framework needs an OS to schedule threads. They support only PThread API and functions they support are create, join, mutex and semaphore - as an addition we support barrier function.

In this study, we proposed XThread Framework, a reconfigurable parallel programming HLS-based library that allows one to design and implement application-specific multi-core hardware for the application of interest from a PThread-like C-level programming to overcome the limitation of the fixed hardware architectures. The importance of this project is to increase the efficiency by using FPGA. Our framework created an application-specific SoC architecture containing many application-specific HLS-based cores that use PThread-like synchronization functions; mutex, barrier, join which are programmed and generated via HLS. XThread consists of the same programming structures in PThread, and it is easy to transfer existing PThread programs in order to speed-up application in reconfigurable low-power and high-performance FPGA platforms.

XThread: A Hardware-based Multi-core Parallel Programming Environment Using Pthread-like Programming Model

XThread framework consists of three parts: (1) reconfigurable core logic, (2) control unit and (3) software application that manages the multi-core framework.

The main contributions of this study as follows;

- Generating highly-optimized core logic in FPGA compared to inefficient compiled code for fixed ISA in CPU.
- Implementing the application-specific cores that work in parallel inside FPGA.
- Employing a PThread-like protocol, handles all synchronization requests and is implemented via HLS on FPGA in order to provide a synchronization mechanism.
- Managing all the data transfer from software to hardware, vice versa, in a PThread-like data communication.
- Running XThread Library Interface we created on any FPGA.

Our responses of Xilinx Open Hardware as follows;

- ***Technical Complexity:*** Our technical challenges were realizing critical section concept in HLS and creating a protocol to provide a communication layer where cores and control unit can talk properly. We achieve the critical section concept by constructing a dependency between XThread synchronization functions and the rest of the code. We have developed an optimized method that will enable the cores and the control unit to communicate correctly.
- ***Implementation, Reusability:*** In this study, we implemented our proposed XThread framework that we propose on KCU105 Xilinx Kintex UltraScale FPGA. In addition, in terms of execution time, this study is easier to implement than a framework at RT-Level. We have proven that our proposed XThread library interface is much more advantageous than the existing PThread library, both in terms of speed and power consumption. Also, XThread can be run on any FPGA by configuring it according to the application to be used. (We already tested it on Zedboard, NEXYS4 DDR).
- ***Marketability/Innovation:*** FPGA-based cloud data center (Amazon Web Services (AWS), Microsoft (Azure), or IBM Cloud) is becoming widespread today and a game changer technology in many different workload accelerations. Providing speed up and low-power consumption for FPGA-based reconfigurable accelerators will be possible with our XThread Library Interface.

2.Background

In this section, we briefly explain what PThread is and how it matters with our study and why we focus on it.

2.1. Introduction

A process is the instance of a computer program that is being executed by one or many threads. The process contains the program code and its activity. A thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Light Weight Processes. Threads all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. Therefore, threads are a popular way to improve application through parallelism and all these threads execute in parallel. The CPU switches rapidly back and forth among the threads giving

the illusion that the threads are running in parallel. There can be multiple threads within a process. Multithreaded programs may have several threads running through different code paths "simultaneously". There are advantages of multi-threaded programming; it utilizes program parallelism on a multiprocessor architecture for faster execution, performance gains on single processor architectures by allowing processes to make computational progress even while waiting on slow I/O operations or blocking operations. Synchronization is needed between different processes or between threads within the same process when they want to access a shared resource provided by the operating system or held by the process itself. Sharing global resources between processes, simultaneous reading and writing on the same variable makes order critical. Shared-memory systems have been developed to eliminate these synchronization problems that may occur. In the shared-memory model, all threads have access to the same global / shared memory and threads have their own private data. So, programmers are responsible for synchronizing access to global memory. Three platform independent shared memory programming models are X3H5[10], Pthread[11], and OpenMP[12].

2.2. Threads in PThread

Portable Operating System Interface for Computer Environments (POSIX) is an interface standard governed by the IEEE and based on UNIX. PThread library is a POSIX C language thread programming interface that has standardized functions for using threads across different platforms. Thus, within POSIX, PThread describes the threading APIs the operating system needs to support. In PThread, threads are general-purpose basic building blocks for multi-threaded applications; it is a low level of abstraction, but it controls over threads explicitly.[14]

Most hardware vendors offer PThread in addition to their proprietary APIs. PThread is defined as a set of C-language programming types and procedure calls. Vendors generally provide users with PThread implementation in the form of a header/container file(***"pthread.h"***) and a library. [11] So that users can easily integrate it into their program.

PThread API can be informally grouped into three major classes: (1) Thread Management, (2) Mutexes and (3) Condition Variables. Thread management functions that work directly on threads, i.e., creating, joining. Mutex functions that deal with synchronization, called a "mutex", which include functions for creating, locking and unlocking mutexes. Condition variables functions that address communications between threads that share a mutex. These are set by programmer specified conditions.

The reasons [13] why we took PThread library as a reference for the XThread model we suggested in our study;

- Light Weight
- Efficient Communications
- Priority/real-time scheduling
- Asynchronous event handling
- Overlapping CPU work with I/O

Also, in order for a program to take these advantages of PThread, it must be able to be organized into discrete, independent tasks that can execute concurrently. In this study, we propose the XThread Library Interface synthesized on HLS, which can be run like PThread library and we provide a hardware-based multi-core parallel programming environment using PThread-like programming model.

3. XThread: A Framework to Design Application-specific Multi-Core Hardware

In this study, we implemented XThread, which is a hardware-based multi-core parallel programming environment using a PThread-like programming model. In this section, we explain how we created XThread framework, how the protocol is being used, what features it has, how it is synchronized and how it is transferred to the hardware.

3.1. Definition and Properties

When trying to solve an algorithm serially, its performance is limited and as a solution to this problem, algorithms are transferred to multi-core programming (OpenMp, PThread, OpenCL) structures. Native POSIX Thread (PThread) library is used for multithreading programming developed in Linux operating system. Through this library, a thread is assigned to a specific core, performance and utilization of a single core increases by using thread-level parallelism. In this library, there are many functions to run threads in parallel and provide necessary synchronization and it is easy to integrate into the algorithm. Each multi-thread programming paradigm basically involves starting and ending threads correctly. In this study, XThread framework proposed, based on the POSIX thread library for multi-core parallel programming integration at the hardware and threads are application specific hardware. While creating XThread framework, it was taken into consideration that users can use this structure with their knowledge without learning any extra programming. In XThread framework, to use the operation provided by the function used in the POSIX -thread library, it is sufficient to write XThread instead of PThread in the corresponding function name. In this study, XThread framework supports some basic functions. Supported functions in XThread framework, their counterparts in PThread library, and descriptions of these functions are shown in Table 1.

Table 1. Supported Functions in the XThread Framework

XThread Framework	PPhread Library	Description
xthread_t thread_id[nThreads]	pthread_t thread_id[nThreads]	Define n different thread types
xthread_create_main()	pthread_create()	Create a new thread that starts execution by invoking
xthread_join_main()	pthread_join()	Wait for the thread specified by thread to terminate
xthread_barrier_init_main()	pthread_barrier_init()	Initialize a barrier
xthread_mutex_init_core()	pthread_mutex_init()	Initialize a mutex
xthread_mutex_t mutex_x	pthread_mutex_t mutex_x	Define a mutex type named mutex_x
xthread_barrier_init_core()	pthread_barrier_init()	Initialize a barrier
xthread_barrier_t barrier_x	pthread_barrier_t barrier_x	Define a barrier type named barrier_x
xthread_mutex_lock()	pthread_mutex_lock()	Lock a mutex
xthread_mutex_unlock()	pthread_mutex_unlock()	Unlock a mutex
xthread_barrier_wait()	pthread_barrier_wait()	Synchronize at a barrier

Core information is kept in the processor using the xthread_t type. By calling create_main() function on the software application, an IP core is started within the SoC architecture. An unique ID is assigned to each core and the addresses of the relevant control unit and memory are sent to the core. On the software application part, all cores on the network are abstracted from each other. This means that while implementing SoC architecture, cores do not need to know each other's addresses. This property provides a flexible SoC design to users. The join_main() function waits for the cores to be finished based on core information in xthread_t. For algorithms containing critical sections, the relevant synchronization function (mutex or barrier) must be defined both within the software application and within the core (HLS) for synchronization management. Note that, XThread supports multiple use of the same synchronization function, but the definition order of the corresponding synchronization function must be the same within the core and software application for the code to work correctly. Thus, mutex_lock(), mutex_unlock() and barrier_wait() functions are supported for IP cores in hardware.

3.2. Protocol

A certain protocol is followed by both control unit and cores while communicating with each other. Firstly, a core tries to assign the control unit to its own process at the beginning of the protocol. More than one core can apply to the control unit at the same time, so the cores need to wait for a signal from the control unit that the assignment was successful. After the core receives the ack-signal as 1 from the control unit, it transfers the parameters for processing. Parameters specify the type of operation (mutex or barrier) and address of the instance of the synchronization mechanism (because a running algorithm can include more than one same synchronization mechanism, for instance, at the same time mutex_1 and mutex_2 may be needed). After parameters are set into the control unit, the core sets go-signal as 1 for the control unit to interpret these parameters. Control unit interprets parameters and then each side of the conversation sets ack-signal and go-signal properly to finish the conversation. At this point core stalls its all operations and waits for ack-signal to be 0. Therefore, the sync mechanism is successfully established. Depending on the synchronization mechanism, at some point where conditions for the core to be awakened are met, the control unit awakes the core by setting ack-signal as 0 and so the core can continue its operation. The talk protocol of XThread scheme is shown in Figure 1.

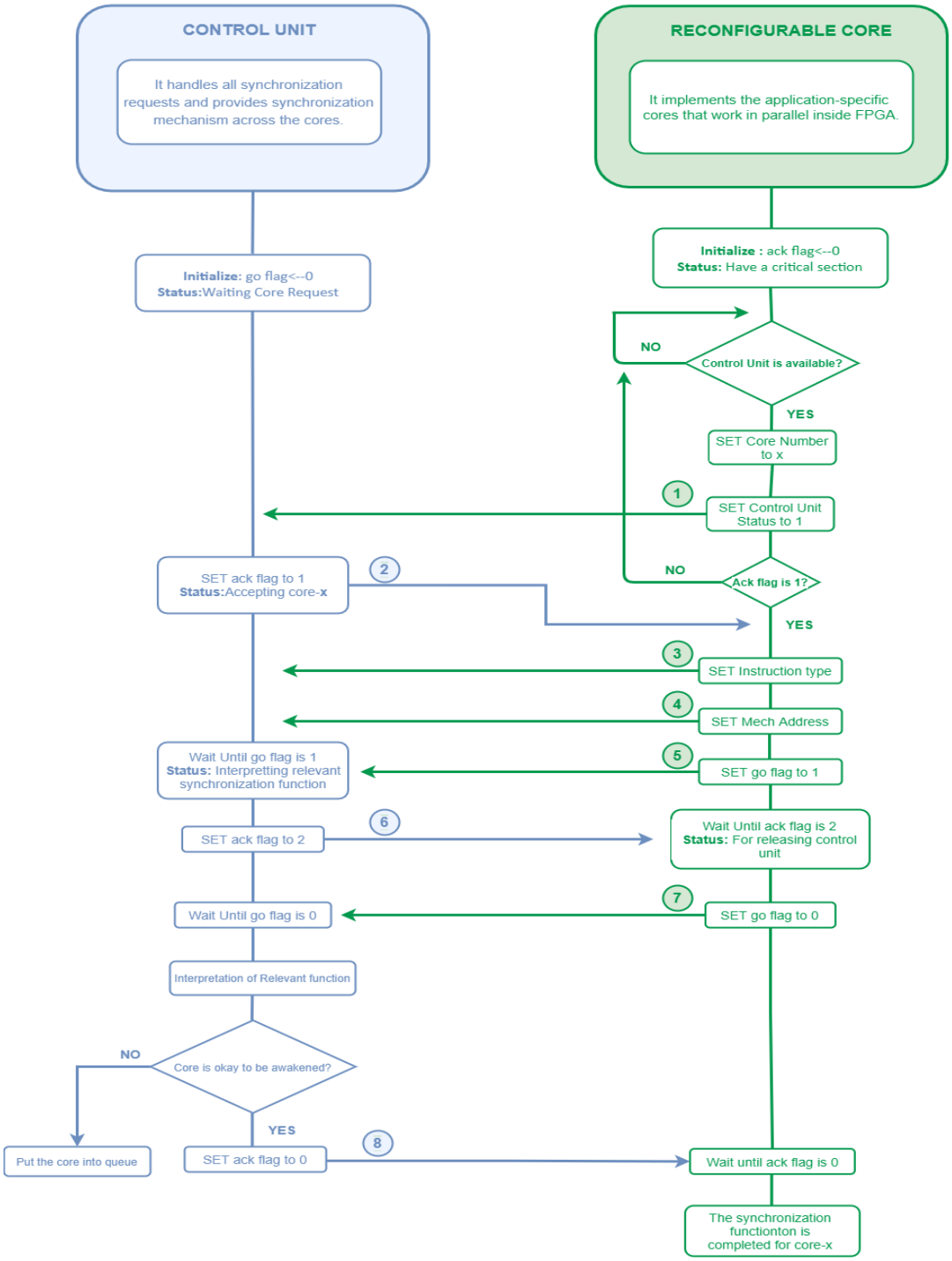


Figure 1. Talk Protocol in XThread

3.3. Synchronization Model

Our framework includes an application-specific SoC architecture containing many application-specific HLS-based cores that use Pthread-like synchronization functions (mutex and barrier), which are programmed and generated via HLS. These synchronization functions allow only certain numbers of cores to work (memory W/R, changing common variables) in the shared-resources. In parallel algorithms, shared

variables may cause dependency issues. In order to overcome these issues, cores need to be synchronized. Several methods are proposed in PThread Library. The synchronization functions supported by the XThread framework are described below.

3.3.1 Mutex

Mutex is a locking and synchronization mechanism. Mutex mechanism allows only one thread can acquire the lock at a time, then enter the critical section and access shared resources. This thread only releases the lock when it exits the critical section. After this process, only one of the other threads can acquire the lock and do its own task. Working mechanism of mutex is shown in Figure 2.

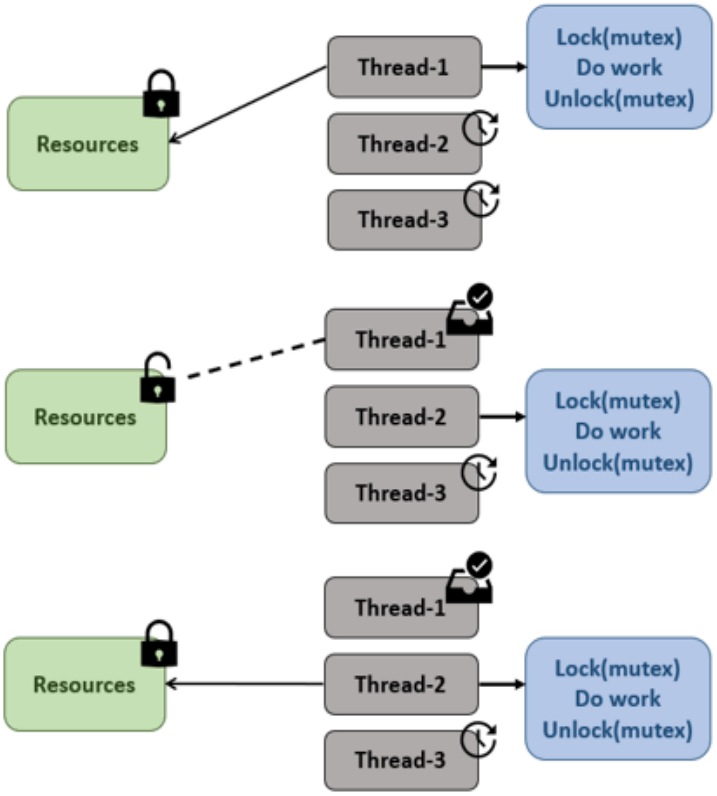


Figure 2. Mutex

3.3.2 Barrier

In cases where you must wait for a number of tasks to be completed before an overall task can proceed, barrier synchronization can be used. In this case, the thread that finishes its task short time must also wait for other threads to finish their work. Barrier is the point in the program where threads stop and wait. When all threads have reached the barrier, they can proceed. Working mechanism of barrier is shown Figure 3.

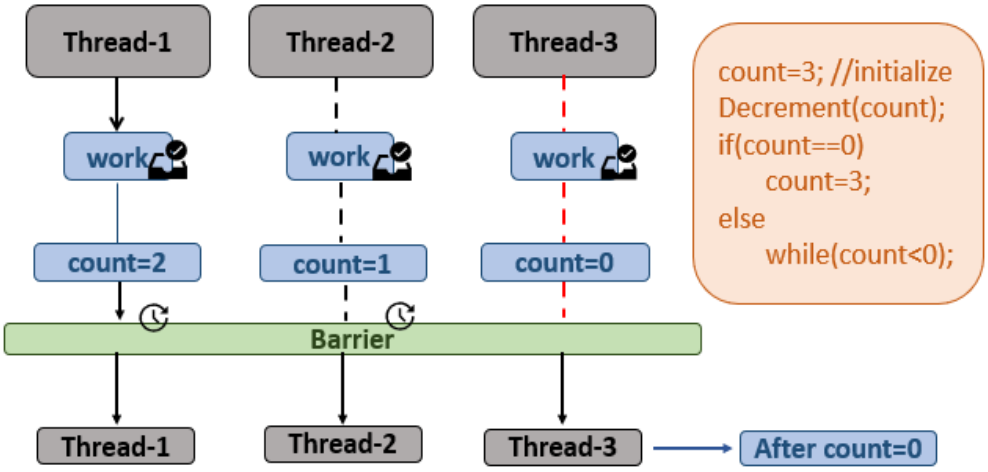


Figure 3. Barrier

3.3.3 Join

Join method blocks the calling thread until the target thread terminates, unless thread has already terminated. Working mechanism of join is shown Figure 4.

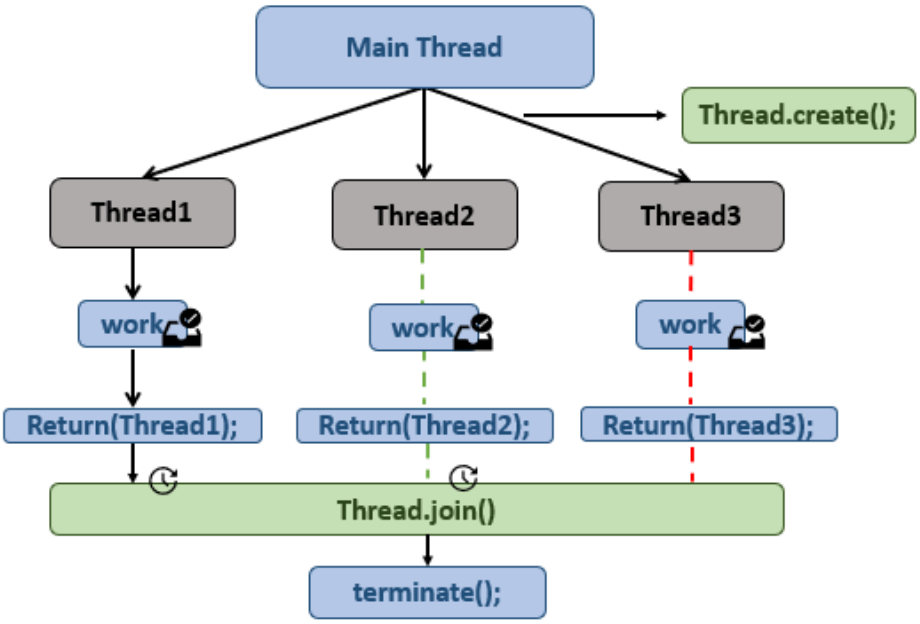


Figure 4. Join

3.4. Implementation

XThread framework consists of three parts: (1) reconfigurable core logic, (2) control unit and (3) software application that manages the multi-core framework. Reconfigurable core logic implements the application-specific cores that work in parallel inside FPGA. Control unit employs a PThread-like protocol,

handles all synchronization requests, and is implemented via HLS in order to provide synchronization mechanisms across the cores. Finally, the software application runs on the software part of the SoC and manages all the data transfer from software to hardware, vice versa, in a PThread-like data communication.

Reconfigurable core side is divided into two parts: algorithm and XThread Library Interface. This part is shown in Figure 5. Implementing a benchmark that is written by using PThread under the XThread framework is quite easy. A PThread code can be separated in two sections: main and runner. In PThread the main function goes directly into the software part of the framework. Runner function is synthesized by HLS. Synchronization functions that are used in the code can be replaced with XThread functions. Thus, the code written using Pthread is transferred to XThread.

Synchronization functions in XThread need some parameters while talking with the control unit. These parameters indicate core number, type of operation and address of the instance of the synchronization mechanism. Also, these functions need addresses of the control unit and the memory unit.

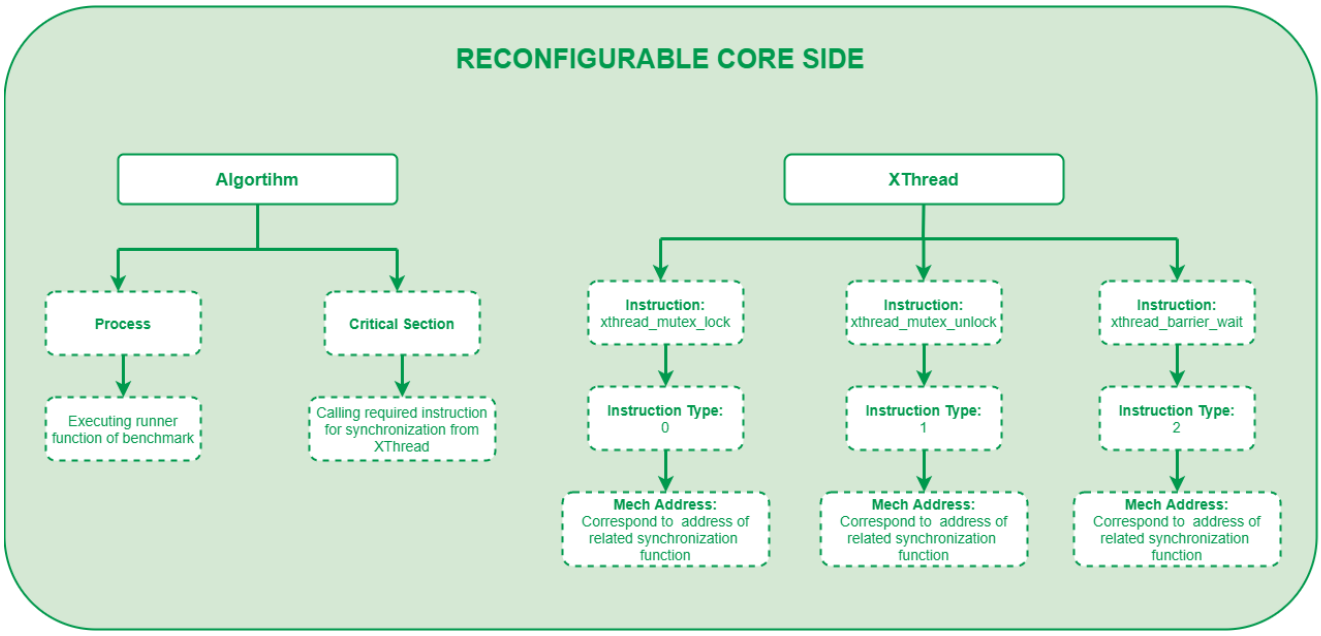


Figure 5. Reconfigurable Core Logic

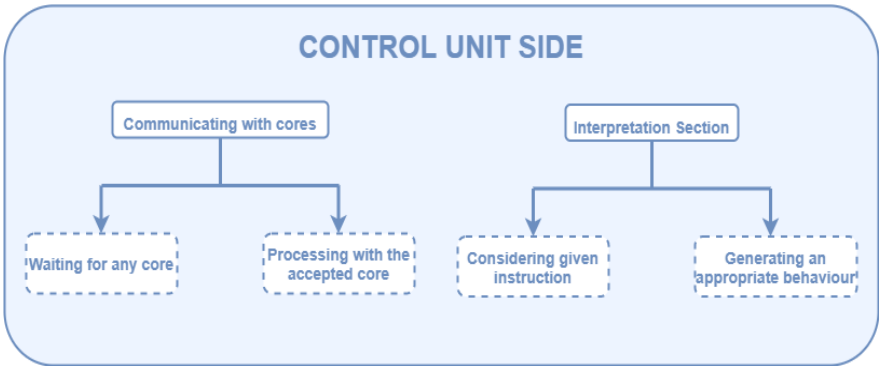


Figure 6. Control Unit

Control unit is a pre-coded non-configurable IP core. It is synthesized before all cores and no need to change it later. In the processing, the control unit waits for cores to apply. It interprets all synchronization functions and behaves like a traffic police to decide which core needs to go and which needs to stall. Behaviors of the control unit are shown in Figure 6.

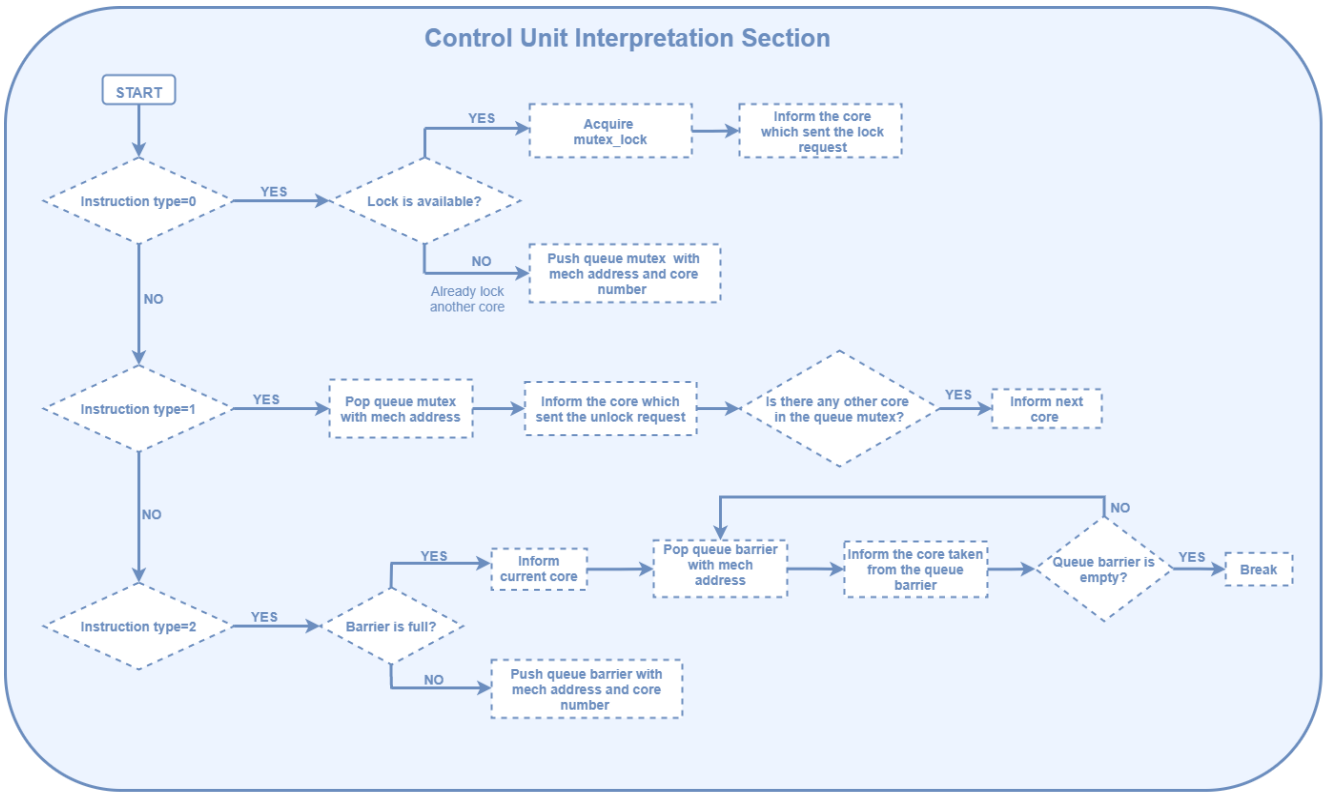


Figure 7. Flowchart of Control Unit Interpretation Part

The control unit behaves differently for each instruction type. This organization is shown in Figure 7.

Instruction type = 0 corresponds to mutex_lock function. At the first it checks whether the given lock is available or already acquired by another core. If the lock is available, then it acquires the lock and informs the core that it can continue its operation. If the lock is already acquired, then the core is put into queue so that it can be awakened later on when the core which acquired the lock releases it.

Instruction type = 1 corresponds to mutex_unlock function. The control unit takes out the core from the given lock and makes it available for any core to acquire. Then inform the core to continue its operation. If there is any core which waits for the lock then the control unit informs it to continue its operation.

Instruction type = 2 corresponds to barrier_wait function. At the first it checks whether the queue for the given barrier is full or not. If the queue is not full, the core is pushed into the queue to be informed later on when the queue is full. If the queue is full then this means the barrier has the maximum predetermined number of cores. At this point the control unit informs all the cores in the queue to continue their operations.

3.5. Reconfigurable SoC Architecture

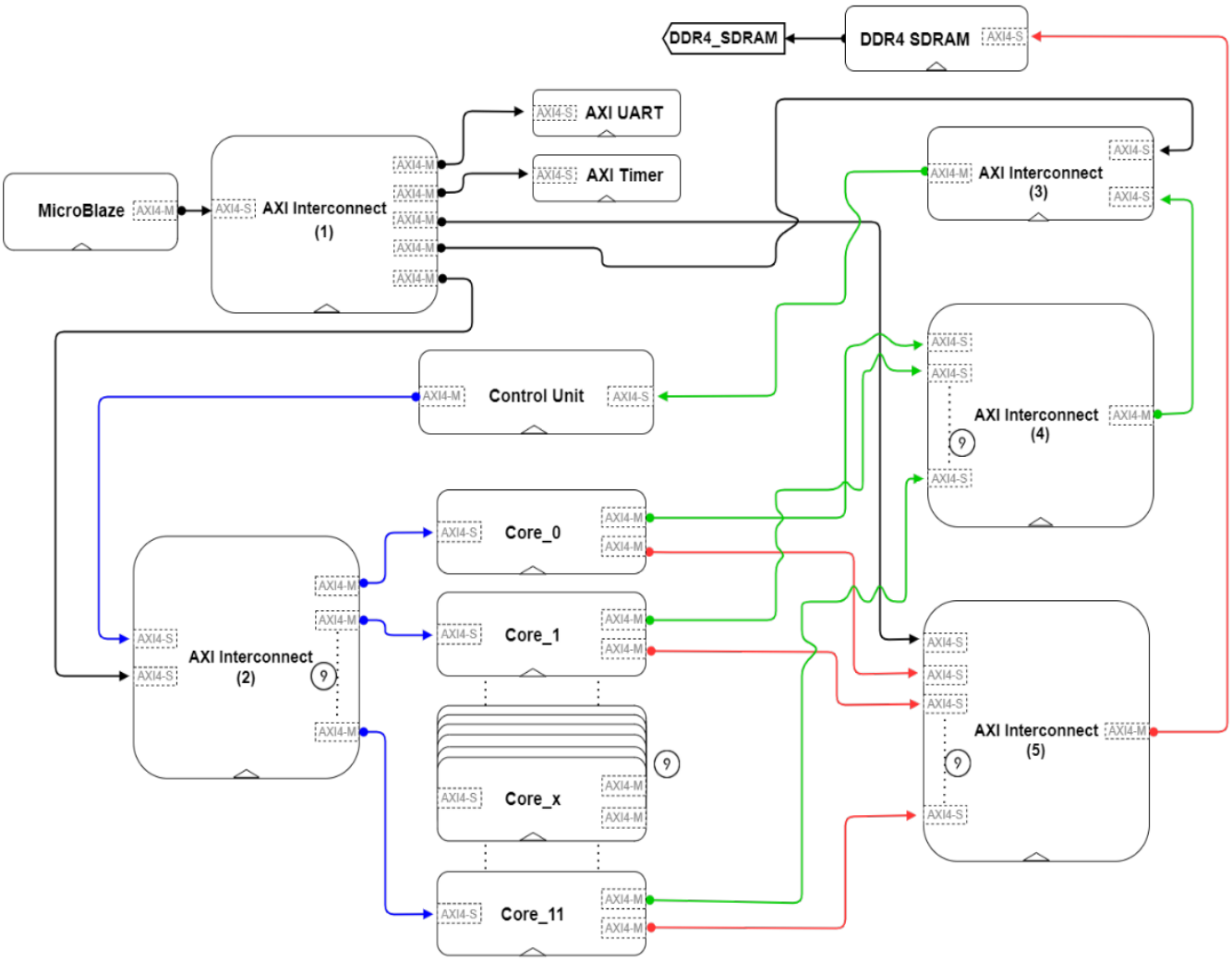


Figure 8. SoC Architecture

The organization of the architecture which involves MicroBlaze and its peripherals, control unit, cores and memory are shown in Figure 8.

MicroBlaze has connections for all IP cores across design. It uses UART to communicate with the computer and Timer to measure the time and it communicates with them through interconnect 1. It connects cores through interconnect 1 and interconnect 2. So, cores can be started by SDK. Connection for the control unit is established through interconnect 1, interconnect 2 and interconnect 3 to initialize some parameters for the control unit. Also, it can write and read memory through interconnect 1 and interconnect 5.

In almost every algorithm, cores need to reach memory for write / read actions. They connect memory through interconnect 5. To use synchronization functions, they need to have a connection for the control unit. This connection is established through interconnect 4 and interconnect 5.

To answer requests from the cores, the control unit has a connection through interconnect 2. So, the control unit can inform cores when the conditions are met.

4.Experiments

In this work, we described and tested a total of three benchmarks. We created two versions of each benchmark, a sequential version of benchmark and a parallelized version of benchmark with XThread. We ran the sequential execution of the algorithm on KCU105 Xilinx Kintex XCKU040-2FFVA1156E UltraScale FPGA only on MicroBlaze. We also tested the parallel version of the algorithm with XThread framework on KCU105 Xilinx Kintex UltraScale FPGA. All tests on the FPGA are performed at a frequency of 100 Mhz. Firstly, we compared and improved these two versions. Then, we compared the performance of the XThread library, which can be used on HLS and SDK we proposed, compared to PThread library. In order to make this comparison, we prepared both the parallel version using PThread library and the sequential version for these three algorithms we chose. The sequential and parallel version ran on a computer using Ubuntu 18.04 version with an Intel i7 7th generation processor on the GCC 6.5 compiler. The operating frequency on Ubuntu is 3800 Mhz.

4.1. Experimental Setup

Implementation of experiment part of this study can be separated into three main parts: (1) HLS configuration that is used to generate all reconfigurable IP cores and control unit, (2) Vivado configuration that is used to create SoC hardware designs and (3) Vitis configuration that is used to verification of software part of SoC design and to communicate data transfer between hardware and software. The process of implementing and testing an experiment involving these three parts is shown in Figure 9.

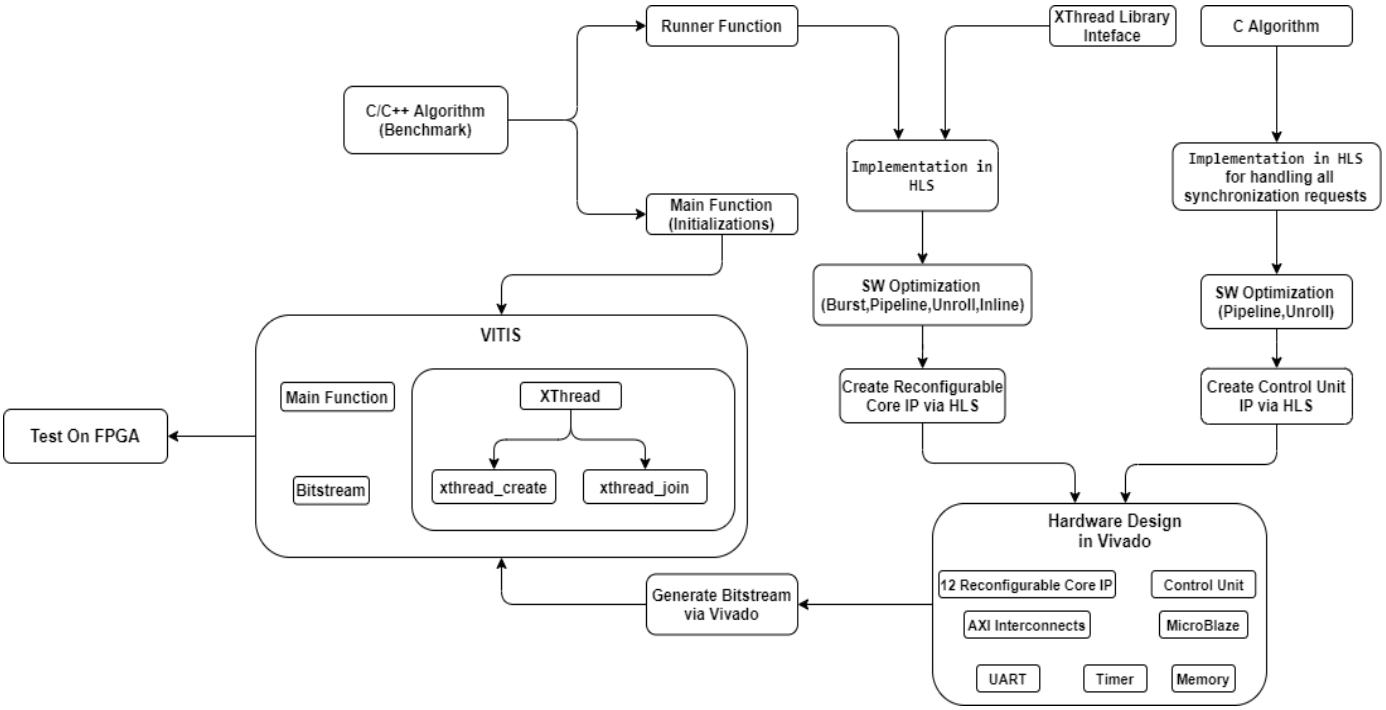


Figure 9. The process of implementing an experiment

4.1.1 HLS Configuration

In this study, the HLS implementation is used to create and package the custom IP reconfigurable core logic and control unit used in Vivado IP Integrator. The control unit is fixed for the XThread framework and provides cross-core synchronization when there is a critical point, and the cores need to talk to each other. By the behavior of the control unit, it decides which core to run, and it wakes them up one by one. Reconfigurable cores are organized and created for each experiment. Reconfigurable core and control unit created via HLS, then exported as RTL as custom core IP to use in hardware. This part is shown in Figure 9. Also, the algorithm is tested on a test bench before creating the custom IP.

In addition, optimizations are applied by adding various pragmas to reduce memory accesses and to speed up benchmarks on the HLS side. The applied HLS pragmas are;

- ***#pragma HLS INTERFACE m_axi port=mem bundle=BUS_MEM max_read_burst_length=16 max_write_burst_length=16:*** Burst mode can provide data from multiple address locations for a single address input. The advantage of this mode is that multiple data can be obtained by providing a single address. In this pragma, the max_read_burst_length and max_write_burst_length parameter option specifies the maximum number of data values read or write during a burst transfer. In this study burst mode is activated and 16 data can be obtained by providing a single address.
- ***#pragma HLS inline:*** This pragma removes a function as a separate entity in the hierarchy. After the inline pragma is applied, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL.
- ***#pragma HLS UNROLL:*** The UNROLL pragma transforms the loops by creating multiple copies of the loop body in the RTL design, allowing some or all loop iterations to happen in parallel.

- ***#pragma HLS array_partition variable=array_x cyclic/block factor=16:*** This pragma splits an array into smaller arrays or individual elements, allowing us to send data to memory faster by creating packets on writes, and complete partitioning decomposes the array into individual elements.
- ***#pragma HLS PIPELINE II = 1:*** This pragma reduces the initiation interval for a function or loop by allowing the concurrent execution of operations. Therefore, the same cycle can be performed in a shorter time.

4.1.2 Vivado Configuration

The SoC architecture setup for the experiment is shown in Figure 8. KCU105 Xilinx Kintex XCKU040-2FFVA1156E UltraScale FPGA is software and hardware programmable. Communication between these two structures is provided through the Advanced Extensible Interface (AXI). Therefore, All communications between the MicroBlaze, memory, core logics, control unit, timer and UART are made through the AXI. Also, Figure 9 is shown which modules are used in the Vivado section. In this study, we implemented multi-core hardware using the XThread library, which we proposed by adding 12 core logic due to resource utilization of the FPGA we use. In this section, it should be noted that the same address is not given when assigning the addresses of cores and the control unit. After the address assignments are made, firstly, the design is synthesized. Then, the implementation of block design is made, and finally the bitstream is created to be tested on the FPGA. In the multi-core hardware design that includes 12 core logics, the time taken for these 3 steps is approximately 2 hours. After 2 hours, the hardware is ready to be tested on FPGA. In some experiments, negative slack may occur while creating reconfigurable core logic. To overcome this negative slack, the Synthesis Setting is arranged Flow-PerfOptimized-high and the Implementation Setting is arranged performance-ExtraTimingOpt in Vivado. Thanks to these settings, ready-to-test hardware is created that does not contain negative slack.

4.1.3 Vitis Configuration

In this part, the created multi core design is run and the performance of the XThread framework is tested on KCU105 Xilinx Kintex UltraScale FPGA. The steps used in the testing process are;

- Including XThread Library
- Determining the addresses of the control unit, core and memory
- Defining as many threads as the number of cores to be used
- Give address of the first core to the control logic
- Preparing the memory and making the necessary definitions for the main function of the benchmark
- Sending needed information to threads with xthread_create_main function
- Wait for the specified threads to terminate with xthread_join_main() function
- Measuring the execution time of the process by increasing the number of cores

In addition, the sequential version of the algorithm used for performance comparison is added as a function and it is measured how long it worked on the FPGA. Also, Figure 9 is shown which modules are used in the Vitis section.

4.2. Benchmarks

In this study, three different benchmarks are tested to observe the performances of the supported synchronization mechanisms. These benchmarks are described below.

4.2.1 Finding Minimum

The Finding Minimum algorithm is dedicated to find minimum value in a number set. Parallel implementation of this algorithm is based on allocating the array between cores and letting cores to find their local minimums and then setting the global minimum. Since the algorithm has a global variable, this global variable may cause misreading. To avoid it, we use mutex feature of XThread. In this benchmark, float data size of 100000000 is used, and the minimum number is found among these data.

4.2.2 Gaussian Elimination

Gaussian elimination is a method used to solve systems of linear equations. The coefficients of the equations are stored in the matrix. And the constant values are on the most right side of the matrix. The algorithm consists of two parts: elimination and back-substitution. In the elimination step, matrix is converted into an upper triangular format. Elimination is performed by starting with the first row and adding a multiple of it to the rows below such that the first column of each of the remaining rows becomes zero. This process is repeated for all of the rows except the last. The selected row that is added to the other rows is called the pivot row. In the back-substitution step, matrix is converted into a form where only constant values and n^{th} element of n^{th} row are in matrix and other cells are filled with zero. So, a variable of systems of linear equations can be found with the way that dividing entire related row by its constant values. Described method above is called as without interchanges. If the matrix has zero in the n^{th} element of the pivot row, the rows need to be swapped for the method to work. This is known as Gaussian elimination with interchanges. The Gaussian with interchanges chooses the unpivoted row with the largest element in the currently selected column. If this row is not the pivot row, it is swapped to become the new pivot row. In this benchmark, we chose size of matrix as 1023*1024 and the data type of that matrix is float.

4.2.3 Block Matrix Multiplication

Blocking is a well-known technique to increase memory usage efficiency. Instead of working on all the rows or columns of an array like standard matrix multiplication, block matrix multiplication performs the multiplication by dividing each matrix to be multiplied into the block matrix that is a matrix divided into blocks that are themselves matrices.[15] Subdivision is performed by vertically and / or horizontally cutting the matrix one or more times. The multiplication of two block matrices is performed using the standard rule for matrix multiplication. With this technique, memory is used more efficiently. In this benchmark, we determined the size of the first and second matrices as 1024* 1024 and we chose block size as 16. The data type of these matrices is float.

4.3. Results and Perspectives

Results of finding minimum on FPGA is shown Table 2. Results of Gaussian Elimination on FPGA is shown in Table 3. Results of Block Matrix Multiplication on FPGA are shown Table 4.

As expected, in these three experiments, the processing time decreased linearly as the number of XThread hardware cores increased. It has been observed that the algorithms are significantly slower than the multicore design when run only on MicroBlaze. In sequential experiments of block matrix multiplication

and gaussian elimination benchmarks, we used smaller sized matrices due to MicroBlaze taking so much time to finish.

Table 2. Results of Finding Minimum on FPGA

Version	Data Size	Number of Core	Time(sec)
Sequential Version	100000000	-	624,957963
Parallel Version with XThread	100000000	1	1,000009
Parallel Version with XThread	100000000	2	0,500012
Parallel Version with XThread	100000000	4	0,250020
Parallel Version with XThread	100000000	8	0,125035
Parallel Version with XThread	100000000	12	0,083384

Table 3. Results of Gaussian Elimination on FPGA

Version	Matrix Size	Number of Core	Time(sec)
Sequential Version	255*256	-	185,3217
Parallel Version with XThread	1023*1024	1	11,647503
Parallel Version with XThread	1023*1024	2	5,851631
Parallel Version with XThread	1023*1024	4	2,946145
Parallel Version with XThread	1023*1024	8	1,487457
Parallel Version with XThread	1022*1024	12	1,014472

Table 4. Results of Block Matrix Multiplication on FPGA

Version	Matrix Size	Number of Core	Time(sec)
Sequential Version	256*256	-	563,317959
Parallel Version with XThread	1024*1024	1	5,127787
Parallel Version with XThread	1024*1024	2	2,691911
Parallel Version with XThread	1024*1024	4	1,317809
Parallel Version with XThread	1024*1024	8	0,670612
Parallel Version with XThread	1024*1024	12	0,513921

We have tested gaussian elimination and finding minimum benchmarks on one core both with XThread and without XThread. We aimed to measure raw needed time for XThread synchronization functions.

In gaussian elimination, we have used a matrix of size 1023*1024 which means barrier_wait function is used 1023 times for one core. And the difference between with XThread and without XThread versions is 0.001464 sec which means in our framework barrier function needs 0.001464 % 2013 = 0.00000143108 sec to be completed in total (According to Table 5). But again only one core is used in this experiment. Therefore traffic in interconnects is at minimum level and there is no queuing time for the core. This result is the raw processing time for barrier function.

In finding minimum, mutex_lock and mutex_unlock functions are used only for once independently from data size. Therefore, Total raw processing time for mutex_lock and mutex_unlock is 0,000004 sec with minimum traffic in interconnects and no queuing time for the core (According to Table 6).

Table 5. Utilization of Barrier Function in XThread Library Interface on Gaussian Elimination Benchmark

Version	Matrix Size	Time(sec)
One core with XThread Library Interface	1023*1024	11,647503
One core without XThread Library Interface	1023*1024	11,646039

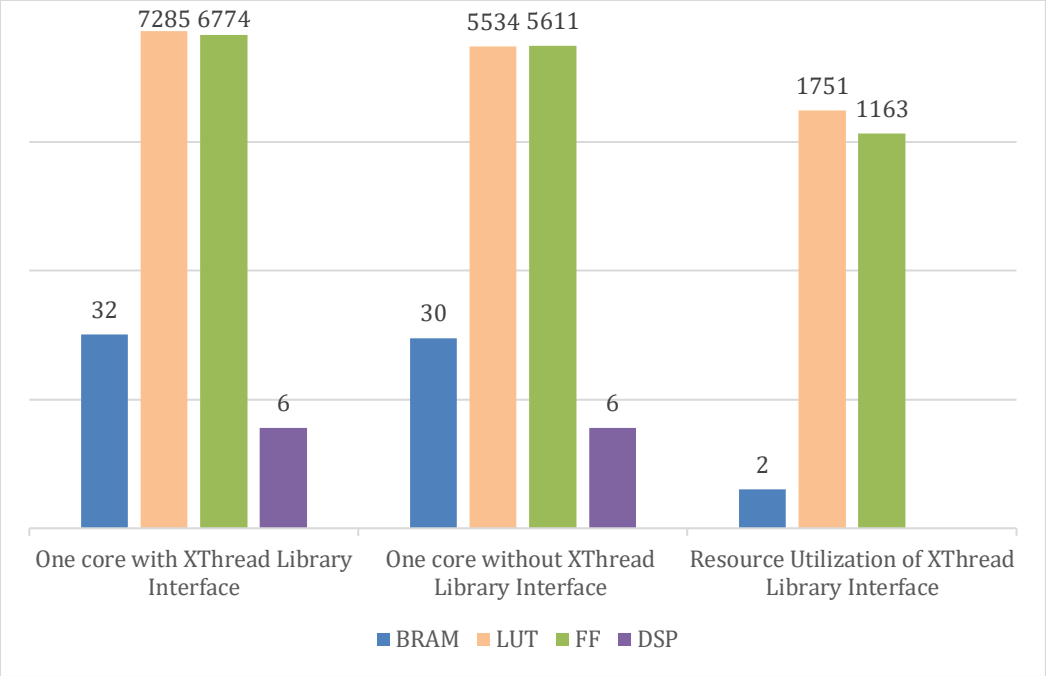
Table 6. Utilization of Mutex Function in XThread Library Interface on Finding Minimum

Version	Data Size	Time(sec)
One core with XThread Library Interface	100000000	1,000009
One core without XThread Library Interface	100000000	1,000005

4.4. Resource Utilization

Resource utilization of XThread library interface is shown in Table 7. According to this table, XThread library interface uses 2 BRAM, 1751 Lookup Table and 1163 Flip-Flop.

Table 7. Resource Utilization of XThread Library Interface



As seen in the resource usage tables, there are 600 BRAM, 242400 Lookup Tables, 484800 Flip-Flops and 1920 DSPs that can be used in the KCU105 Xilinx Kintex UltraScale FPGA. Resource utilization of finding minimum is shown Table 8, resource utilization of gaussian elimination is shown Table 9 and resource utilization of block matrix multiplication is shown Table 10.

Table 8. Resource Utilization of Finding Minimum

Resource	Available	HLS (One Core)	HLS (Control Unit)	SoC Architecture
BRAM	600	32	9	467.5
LUT	242400	7285	4349	109276
FF	484800	6774	2197	195235
DSP	1920	6	0	75

Table 9. Resource Utilization of Gaussian Elimination

Resource	Available	HLS (One Core)	HLS (Control Unit)	SoC Architecture
BRAM	600	36	9	491.5
LUT	242400	9608	4349	139021
FF	484800	11689	2197	216756
DSP	1920	14	0	171

Table 10. Resource Utilization of Block Matrix Multiplication

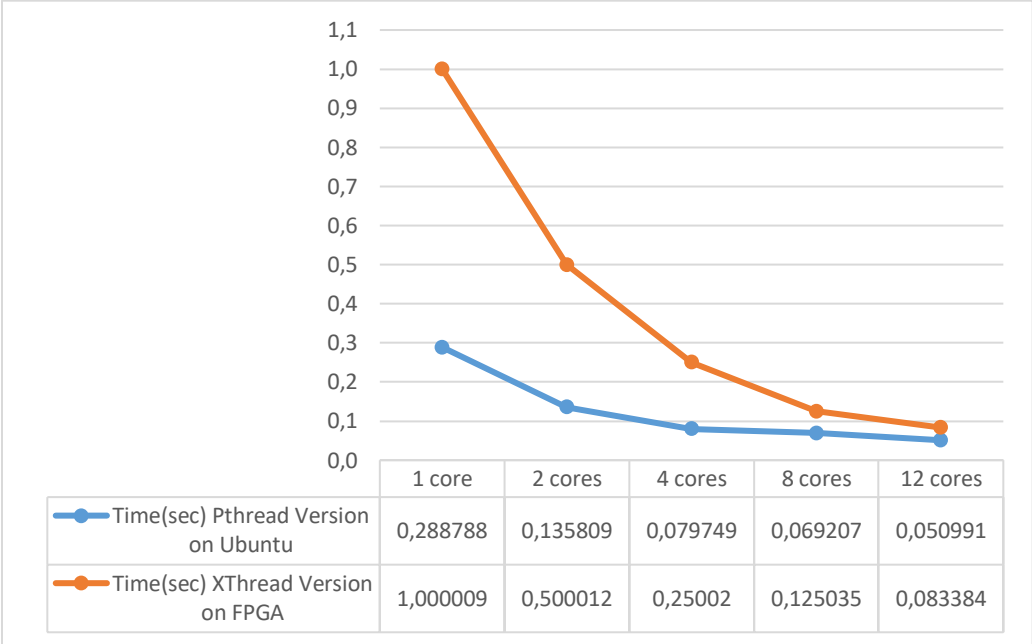
Resource	Available	HLS (One Core)	HLS (Control Unit)	SoC Architecture
BRAM	600	38	9	503.5
LUT	242400	10647	4349	157425
FF	484800	10911	2197	232365
DSP	1920	36	0	435

4.5. Performance Comparison of PThread on CPU vs XThread on FPGA

In this section, we compared the performance of PThread library and XThread library interface we proposed, depending on the number of cores. In these experiments, it is worth highlighting that benchmarks are tested under about 3800 MHz for computer and 100 MHz for FPGA. Despite this 38 times difference, it is recorded that XThread has better results than PThread.

In the finding minimum experiment, the PThread library is approximately 1.64 times faster than the XThread library interface when the number of cores is 12. Considering the frequency difference between them, the XThread library interface is much more advantageous. And also, It provides less power consumption.

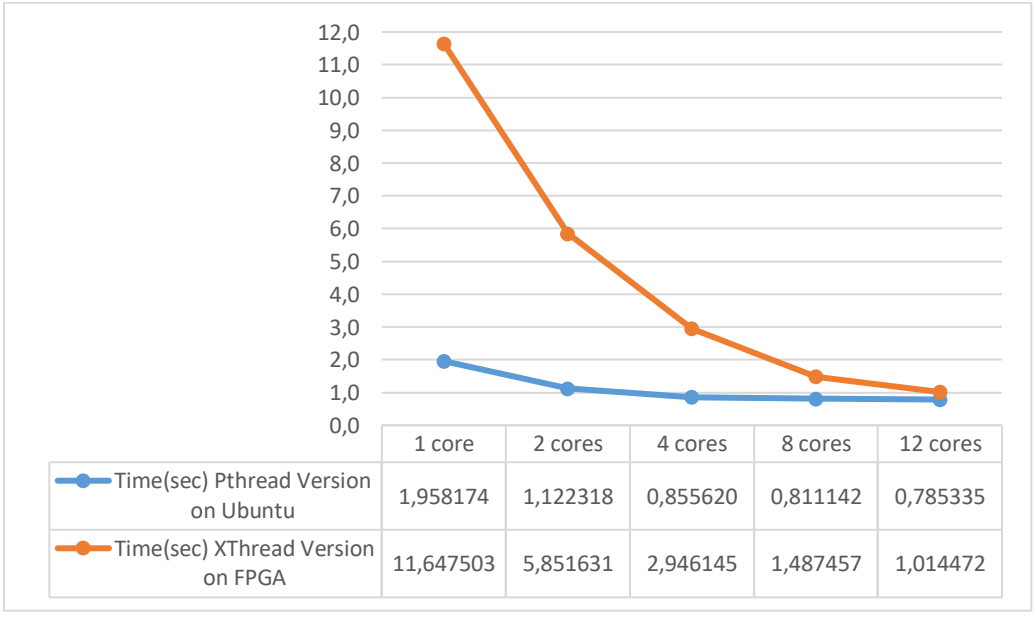
Table 11. Comparison of PThread and XThread Results for Finding Minimum Benchmark



	1 core	2 cores	4 cores	8 cores	12 cores
Time(sec) Pthread Version on Ubuntu	0,288788	0,135809	0,079749	0,069207	0,050991
Time(sec) XThread Version on FPGA	1,000009	0,500012	0,25002	0,125035	0,083384

In the gaussian elimination benchmark, the PThread library is approximately 1.29 times faster than the XThread library interface when the number of cores is 12. In the same way, considering the frequency difference between them, the XThread library interface is much more advantageous. And also, it provides less power consumption.

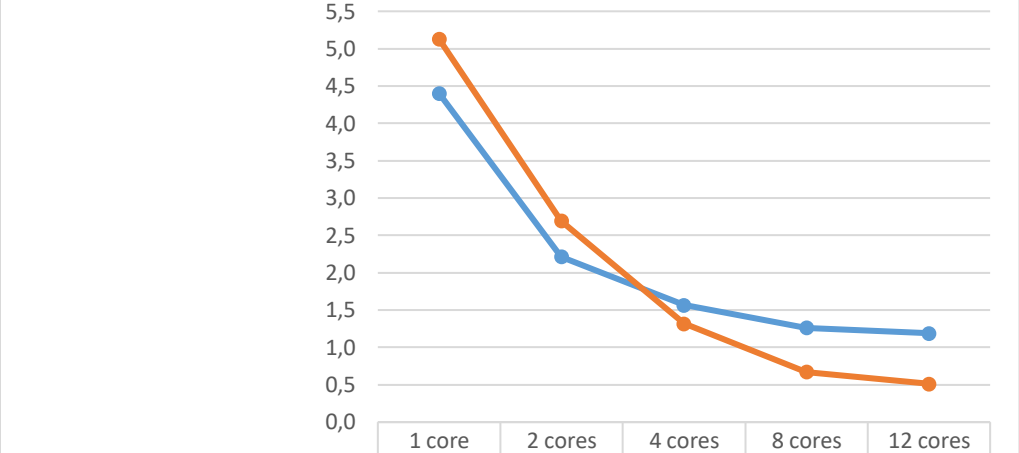
Table 12. Comparison of Pthread and XThread Results for Gaussian Elimination Benchmark



	1 core	2 cores	4 cores	8 cores	12 cores
Time(sec) Pthread Version on Ubuntu	1,958174	1,122318	0,855620	0,811142	0,785335
Time(sec) XThread Version on FPGA	11,647503	5,851631	2,946145	1,487457	1,014472

In the Block Matrix Multiplication experiment, while the number of cores is 12, the XThread library is about 2.32 times faster than the PThread library. When calculated with the frequency difference between them, XThread hardware is very advantageous in terms of both speed and low power consumption.

Table 13. Comparison of PThread and XThread Results for Block Matrix Multiplication Benchmark



	1 core	2 cores	4 cores	8 cores	12 cores
Time(sec) Pthread Version on Ubuntu	4,401710	2,216060	1,568645	1,261123	1,19074
Time(sec) XThread Version on FPGA	5,127787	2,691911	1,317809	0,670612	0,513921

5.Conclusion and Future Work

In this work, we presented a XThread Library Interface which is a hardware-based multi-core parallel programming environment using Pthread-like programming threads to parallel hardware accelerators. XThread is used for hardware accelerators which execute concurrently in a shared memory system. Different synchronization mechanisms that are mutex, barrier and join were provided for XThread Library Interface.

HLS is a powerful design automation tool that automatically translates computations described via high-level programming languages, e.g., OpenCL, C, and C++ into very low-level hardware description languages. In the near future, its importance will have more significance due to increased complexity of digital design space in a single silicon chip. Automation of a very efficient digital design approach is a current and a future research problem. Our approach presented in this project will help automating the synthesis of digital systems by automatically reducing latency, improving performance, and reducing chip area by utilizing multi-core programming advantages. Using the PThread library on the Intel processor, each experiment was run on a different number of cores and the execution times were recorded. These experiments were run on different numbers of cores with optimized XThread hardware in The Kintex® UltraScale ™ FPGA KCU105 and execution times were recorded. Comparing the results of Intel processor and FPGA experiments, XThread Library Interface provides the user with less power consumption in all applications. Considering the frequency difference of Intel processor and FPGA, XThread hardware is much more advantageous than PThread hardware in all experiments. In the finding minimum experiment, the PThread library is approximately 1.64 times faster than the XThread library interface when the number of cores is 12. In the gaussian elimination benchmark, the PThread library is approximately 1.29 times faster

than the XThread library interface when the number of cores is 12. In the Block Matrix Multiplication experiment, while the number of cores is 12, the XThread library is about 2.32 times faster than the PThread library. Note that, despite more than 38 times frequency difference, it is recorded that XThread is faster than PThread in three different benchmarks. Therefore, our proposed XThread is much more advantageous in terms of both speed and power consumption.

The following features can improve and accelerate the XThread Library Interface we proposed:

- PThread runner function can take arguments to be used in the process of calculation. Cores in XThread can not be initialized with arguments. But we have a workaround for this case. MicroBlaze can initialize memory, so cores can read this before the actual process. We intend to include these steps into XThread. Thus, users can easily give arguments into cores.
- PThread makes it possible for threads to wait for other threads. This makes implementing more complex algorithms possible. For example, merge-sort is a far known example of this usage. XThread supports join function only for Vitis. We intend to implement join method for the reconfigurable core side of our framework.
- The method that XThread join function uses is checking whether the core has done signal or not. This polling method causes traffic in related interconnects and this may cause a delay when the control unit needs to inform cores.
- Even if we have accelerated computing with the proposed XThread Library Interface. Memory access, especially to off-chip memory access, is the bottleneck in many application fields, e.g., memory-intensive applications such as neural networks, matrix operations, and graph processing. Memory hierarchy with different cache levels is the solution to the memory access problem, and it will reduce the overall memory access latency of the application. Memory access in our study can be improved by adding cache structure to reconfigurable core logic.
- XThread supports PThread-like semaphore structure. It is tested and well worked. It has not yet been tested on any benchmark to compare results. So, we do not have an analysis on its contribution.

References

[1] A. Canis et al. 2013. LegUp: An Open-source High-Level Synthesis Tool for FPGA-based Processor/Accelerator Systems. TECS 13, 2 (2013).

[2] Xilinx Vivado HLS. 2017. (2017). https://www.xilinx.com/

[3] Intel HLS Compiler. 2017. (2017). https://www.altera.com/

[4] V.G. Castellana, A. Tumeo and F. Ferrandi. 2014. High-level Synthesis of Memory Bound and Irregular Parallel Applications with Bambu. In HCS. IEEE, Cupertino, CA, USA.

[5] J. Choi, S. Brown, and J. Anderson, “From software threads to parallel hardware in high-level synthesis for fpgas,” in 2013 International Conference on Field-Programmable Technology (FPT), 2013, pp. 270– 277.

[6] J. Cheng, S. T. Fleming, Y. T. Chen, J. H. Anderson, and G. A. Constantinides, “Easy: Efficient arbiter synthesis from multi-threaded code,” in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2019, pp. 142–151.

[7] Y. Y. Leow, C. y. Ng, and W. f. Wong, “Generating hardware from openmp programs,” in 2006 IEEE International Conference on Field Programmable Technology, 2006, pp. 73–80

[8] F. Vahid, G. Stitt, and R. Lysecky, “Warp processing: Dynamic translation of binaries to fpga circuits,” Computer, vol. 41, no. 7, pp. 40–46, 2008.

[9] G. Stitt and F. Vahid, “Thread warping: A framework for dynamic synthesis of thread accelerators,” in 2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007, pp. 93–98

[10] W. G. Rudd, “X3h5 parallel extensions for programming language c,” USA, Tech. Rep., 1993.

[11] B. Nichols, D. Buttlar, and J. P. Farrell, Pthreads programming. O’Reilly & Associates, Inc., 1996

[12] L. Dagum and R. Menon, “Openmp: an industry standard api for shared-memory programming,” IEEE computational science and engineering, vol. 5, no. 1, pp. 46–55, 1998.

[13] LLNL HPC Tutorial. https://hpc-tutorials.llnl.gov/

[14] A. Castelló, R. M. Gual, S. Seo, P. Balaji, E. S. Quintana-Orti, and A. J. Pena, “Analysis of threading libraries for high performance computing,” IEEE Transactions on Computers, vol. 69, no. 9, pp. 1279–1292, 2020

[15] M. D. Lam, E. E. Rothberg, and M. E. Wolf, “The cache performance and optimizations of blocked algorithms,” ACM SIGOPS Operating Systems Review, vol. 25, no. Special Issue, pp. 63–74, 1991.s