An FPGA-Based Computation Model for Blocked Algorithms

YI-GANG TAI, CHIA-TIEN DAN LO, and KLEANTHIS PSARRIS

Department of Computer Science
University of Texas at San Antonio
6900 N. Loop 1604 West, San Antonio, TX 78249
U.S.A.

Abstract: - Long running applications typically involves huge amount of iterations of loops with or without loop dependencies over a set of data that may not be loaded to memory as a whole. In this paper, an FPGA-based computation model is proposed to optimize this type of application that requires extensively load/store a block of data in a loop by utilizing the flexibility of the FPGA-based re-configurable architecture. The kernel of the computation model is a set of computation cores, each of which consists of three pipeline stages and dual buffers to shorten memory access latency. Multiple cores can be configured to operate over a set of data in a buffer, similar to the traditional multiple instructions with single data stream (SIMD) concept. Preliminary results in terms of representative examples such as pattern matching algorithms and QR matrix decomposition algorithms are presented.

Key-Words: FPGA, Reconfigurable Computing, Blocked Algorithms, QR Decomposition, Loop Optimization

1 Introduction

Reconfigurable hardware fills the gap between application specific integrated circuits (ASIC) and general-purpose processors. ASICs usually provide the highest performance because of the optimization done by the designer. However, it requires considerable design efforts and the ASIC chip designed for a desired application generally cannot be used for other applications. In addition, some applications are very difficult to be implemented in ASICs. On the other hand, for a general-purpose processor, flexibility is one of the most important design considerations in order to run any applications. However, the performance running a specific application is often limited by its general-purpose architecture.

Field programmable gate arrays (FPGA) are the most common reconfigurable hardware. A typical FPGA consists of look-up tables (LUT) and special functional units, which are surrounded by a matrix of programmable interconnects and programmable I/O units. LUTs can be configured to perform basic digital logics and interconnects are also configured in a way the whole piece of hardware is realized in combinational or sequential circuits.

Reconfigurable systems can be classified into two basic categories according to the time hardware configurations are configured. The first category is called compile-time reconfigurable systems. The reconfigurable hardware is configured at power up time and remains unchanged. The hardware configuration has to be built prior to configuring hardware and the hardware performs fixed functions. In the second category, called run-time reconfigurable systems, the configuration of the reconfigurable hardware can be changed during runtime. Hardware configurations, most of them precompiled, can be loaded into FPGAs when they are needed by program execution.

On the other hand, it is sometimes desirable to compile hardware configurations at runtime. If hardware configurations are compiled at runtime based on execution statistics, arbitrary programs can be analyzed and accelerated if applicable. Similar situations can be found in virtual machines (VM) of interpreted languages such as Java. Just-in-time (JIT) compilation is one of the techniques that compile
virtual machine instructions to native code at runtime to further improve execution performance. Newer JIT compilers dynamically select a subset of program hot spots for compilation and optimization based on program execution statistics. Similar concept could possibly be applied to FPGA configuration compilation.

However, in typical circumstances, building a complete hardware configuration involves a long design flow, which includes stages of synthesize, translate, map, place and route, and finally the generation of configuration bit stream. The time these stages consume makes the traditional design flow unsuitable for runtime compilation. In recent technologies, partial reconfiguration provides a way to configure part of the device while the other part remains working. In this way only a small part of the whole design has to be built at a time and the building time is shortened.

Partial reconfiguration requires modular design architecture so the device can still do its job while a part of it is being reconfigured. In this paper we propose a modular design architecture targeting on algorithms manipulating large amount of data, such as operations on large matrices. The size of the matrices involved is large enough that they cannot be entirely loaded into FPGA internal memory or typical cache in a host processor. Traditional linear algebra implementations use blocking (tiling) to improve their performance. Blocked algorithms improve data reuse in cache memory, hence the performance gain. In our proposed architecture, block buffers are designed for this purpose for blocked algorithms.

Blocking benefits many algorithms. QR decomposition of matrices is one of the important mathematical tools for solving linear equations. For example, the non-negative least squares (NNLS) algorithm [1], [5], a versatile algorithm from Lawson and Hanson, contains a secondary loop that solves unconstrained least squares problems repeatedly within the main loop, and these unconstrained least squares are often solved using QR decomposition. Applications that employ NNLS often concern large matrices, and this is where blocking benefits most.

Traditionally QR decomposition of large matrices is based on the Householder transformation. In the LAPACK library [3] [1], blocked routines are provided for high performance QR decomposition. However, in these routines, blocking is performed in blocks of full columns of matrices. In other words, if the number of rows of matrices is too large, it may not be possible to bring an entire column into cache; the approach is not very scalable. Square tiling of QR factorization [4] solves this problem, at the cost of applying an update routine. The algorithm is developed for out-of-core applications, i.e. the majority of matrix data is stored in disks. On the other hand, Yi et al. proposed compiler techniques for automatic blocking of QR factorizations [5] into blocks of columns. Automatic blocking is desired for our purpose, yet techniques of automatic blocking into square tiles are still unclear.

The rest of this paper is organized as follows. The next section describes the proposed computation model, which decouples the computation engine from the core controller, and utilizes buffers to improve data reuse. In Section 3, we discuss data path in the computation model since the data in our target application are mainly stored in a memory subsystem that is accessed through a system bus. Section 4 shows the experimental results on the proposed computation model. We conclude this paper in Section 5.

2 The Proposed FPGA-Based Computation Model

The proposed FPGA-based computation model is shown in Figure 1 in which the system data bus can be synthesized using CoreConnect processor local bus (PLB) [10]. Modules below the system data bus form a memory subsystem, including a memory controller and memory modules. The right part of the dash line in Figure 1 consists of a computation engine, which is
composed of computation cores and buffers. The computation core can be reconfigured based on computing paradigms of an application. For example, the computations can be configured to floating point multipliers if an application needs a lot of floating point multiplications.

The left part to the dash line in Figure 1 depicts a core controller which is designed in accordance to the underlying memory subsystem. The core controller consists of a scheduler and a load/store module, which are fixed design. The scheduler generates control signals and activates the computation cores in order and coordinates the buffers based the computation patterns of an application to be executed. This load/store module receives control signals from the scheduler, and update data in the buffers accordingly.

Each of the computation cores implements a specific computation in hardware, e.g., a floating point algorithmic or a string matcher. The controller in a computation core controls the access to the buffers. Typically array indices are used to retrieve the array stored in the buffers. The computing entity in the computation core is the datapath of the computation. Application targets that benefit from hardware implementations are loops with large amount of iterations and computations in the loop body. The computation core can be pipelined or non-pipelined, depending on the application. Pipelined implementations increase clock rate and throughput; however loop-carried dependences stall the pipeline. For loops with dependences, a non-pipelined approach can be used to minimize the latency of a single iteration, with feedback path to reduce the number of memory references [9]. Whether the implementation is pipelined or non-pipelined, the adequate clock rate for the computation core depends on the specific application. Clock rates vary from application to application and may be very different from the (memory) system clock rate. Thus the buffers have to be able to adapt to different clock rate.

The buffers load application data from external memory and provide fast access to these data to the computation core. Unlike caches, however, buffer data replacement is determined by application algorithms. With a duplicate buffer for each of the buffers, data are loaded in background when its original copy is busy and, thus, external memory access latency can be hidden from the computation core. The number of the buffers and the size of each of them are also determined by application algorithms, limited to available FPGA resource. Like buffers, the computation cores have multiple editions in Fig. 1. There are cases, for example, loops that are not perfectly nested, where different computations are to be applied on the same data set. The computation core can be reconfigured to adapt to another computation.

The core controller coordinates the overall operation of the design. It generates addresses both to the buffers and the underlying memory subsystem with correct bus signals and timing. It also has to initiate background loading/storing from/to the memory subsystem. Parameters, needed for sequentially loading/storing sub-matrices (blocks) in the memory subsystem, are simply block size, total number of blocks, matrix dimensions, etc. The next block of a matrix block can be loaded into the inactive buffer of a buffer pair and thus the buffer I/O latency can be shortened.

3 Data Path in the Computation Model

For applications such as the blocked algorithm studied in this paper, the large volumes of data are stored in the memory subsystem, e.g., DRAM in particular. A block of data has to be loaded to the buffers before the computation starts. To improve overall performance, it is important to shorten data path latency while data hit ratio can be increased. In our scheme, a block of data is cached in the buffers after the first reference. This allows multiple computation cores to operate on the same data set. Furthermore, each buffer can hold two blocks of data. In case of a buffer miss, the referenced block can be loaded in the background.

A buffer design proposed by Park et al. employs FIFOs to feed data from the memory sub-system to the computation cores [5] and [7]. An external memory access controller was designed for an FPGA-based reconfigurable computing engine targeting at signal processing applications, in which data access is in the form of streams. Therefore, data addressing has to be computed beforehand and data ordering is crucial to the computation core. Because data are read form the FIFOs, the same data set can not be reused in another computation core. In the above example that multiple computation cores are operating on a data set, both the addressing and the computation cores have to be reconfigured. However, the space taken for the FIFOs is smaller than the buffers used in our architecture.

Our target applications of blocked algorithms with large amount of matrix data bring different design
considerations. Data access may not be easily abstracted into streamed pattern and true loop-carried dependences may not be infrequent, except some applications in signal processing [3]. Besides, there are often multiple different computation cores operating on a block of data. Therefore, caching data into randomly accessible internal memory may be more beneficial than into FIFOs. Furthermore, if multiple buffers to hold blocks of data are used, overlapping memory I/O with core computation can hide various DRAM access latencies.

4 Results
In this section, preliminary results are presented. First the execution model is described, and then examples of various types of loops from different applications are considered.

4.1 Execution Model
With the FPGA design architecture proposed in Section 3, the execution of the design can be divided into three phases: 1) loading, 2) execution, and 3) storing. During the loading phase, the core controller selects buffers and computation cores for a computation to be executed. Then it arbitrates the system bus and data are fetched from the memory subsystem to the selected buffers. After the data are loaded, the computation core executes the operations. The execution can be pipelined or non-pipelined, according to the algorithm implemented. During the storing phase, core controller again transfers computation results from the buffers back to the memory subsystem. The operations in these three phases can be parallelized. For example, while a buffer is executing in a computation core, the other can be loading another block of data. Longer execution phase can hide loading/storing latency better.

Illustrated in Fig. 2 is a simplified execution model that can be use to evaluate the performance of this design. Block 1 denotes the first set of blocks of data to be loaded, and Block 2 is the operations on the next set of blocks after the first set of blocks. In the loading phase, data are fetched from external memory to block buffers, and \( L_{RD} \) denotes the latency. Consider the case of typical SDRAM with full page burst, read latency consists of ACTIVE command to READ command delay, CAS latency, and the system bus / memory controller delay, which is assumed to be a constant if there is no bus interference. Meanwhile, \( T_{RD} \) denotes the data transfer time. When burst transfer is used, it costs one cycle for each location read, so \( T_{RD} = N \) assuming \( N \) locations. For typical SDRAM operating at 100 MHz with IBM PLB as system bus, \( L_{RD} \) is about 9 cycles and the loading phase takes 9+\( N \) cycles to load one buffer, provided that \( N \) is smaller than the column size of the SDRAM, and the operation does not cross a row boundary. If, for example, a 1K x 64 block buffer is to be loaded, with 9-bit column address SDRAM as counterpart, at least two rows have to be activated and one row boundary has to be crossed, and \( L_{RD} \) is increased accordingly. It is similar to the write latency \( L_{WR} \) at the storing phase, except that there is no CAS latency and the write delay is shorter at the system bus / memory controller. In the same scenario \( L_{WR} \) is about 5 cycles.

4.2 Example Cases
The first case to be considered is the inner-most loop of a pattern matcher where three independent arrays are accessed and there is no loop-carried dependence, and hence the loop can be easily pipelined [13]. Fig. 3(a) illustrates the code. There is another loop with index \( i \) that brackets the code shown, but here only the inner-most \( j \) loop is considered. Before the loop starts execution, the data in \( pat \) and \( x \) arrays have to be loaded from the memory subsystem into the buffers containing the arrays. In the cases where these arrays are not too large, they may be loaded entirely into the buffers. For the scenario demonstrated in section 4.1, the timing diagram is shown in Fig. 3(b). Operation time for each block is evaluated to be 4 cycles. The entire arrays are loaded at the beginning, and execution and storing phases follow. Execution time of each iteration in the execution phase is assumed to take 4 cycles.
Fig. 3(c) shows the case that the arrays are too large to fit into the buffers, where $N_B$ denotes the block size. Note that the loading phase of Block 2 is overlapped with the execution phase of Block 1. However, if memory subsystem access is to be entirely hidden, the time of loading has to be less than or equal to the time of execution, and this is not possible in this special case. In fact, the loading of more than one block buffer can not be totally hidden for larger block size for fully pipelined loops if the computation core and memory subsystem interface are working at the same clock rate, but a good portion of memory access can still be hidden, depending on how many buffers to be loaded.

Fig. 4(a) shows another example of a simplified inner-most loop from a blocked algorithm of QR decomposition [10]. The index $j_2$ is loop invariant in this inner-most loop, and thus there is loop-carried dependence at $w[j_2]$ across different iterations. For a straightforward implementation, the loop cannot be pipelined because of the dependence. However, in this special case, it can be improved by adding $p$ registers for temporarily storing $w[j_2]$ in each iteration, where $p$ is the depth of the loop pipeline implemented. In this way, the actual write to the “cached” $w[j_2]$ in the buffer is done in the last stage of the loop pipeline. Note in this example that array $a$ is read twice in the statement, but is not written. Another possible improvement by using this architecture is the buffer that holds the array can be loaded by the way it is used. Column $j_2$ and column $i$ in $a[j_1,j_2]$ and $a[j_1,i]$ respectively can be viewed as different source and loaded into different buffer for faster computation core access. The execution model of the above scenario is illustrated in Fig. 4(b). Because of the dependence, the execution in each iteration cannot be pipelined, hence the non-overlapped $T_{EX}$ in the figure. Moreover, since $w[j_2]$ is temporarily stored in the computation core, it only has to be loaded once in the very beginning and stored once at the end.

**5 Conclusion**

In this paper, an FPGA-based computation model for implementing algorithms into hardware is proposed. The architecture provides interfaces to access the memory subsystem through buffers. These buffers keep partial data of a large entire data set stored in the memory subsystem and provide fast access for the computation core. The computation core in the proposed architecture is an independent module that can be reconfigured without modifying other fixed core controller modules. In the future, this architecture will be implemented and work closely with a
processor for the purpose of online hardware acceleration of software applications. Additionally, virtual memory and cache coherence problems with the host processor are to be studied.

References: