Parallel FPGA-based All-Pairs Shortest-Paths in a Directed Graph

Uday Bondhugula[1], Ananth Devulapalli[2], Joseph Fernando[2], Pete Wyckoff[3], and P. Sadayappan[1]

[1]Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{bondhugu, saday}@cse.ohio-state.edu

[2]Ohio Supercomputer Center (Springfield)
1 South Limestone St., Suite 310
Springfield, OH 45502
{ananth, fernando}@osc.edu

[3]Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

With rapid advances in VLSI technology, Field Programmable Gate Arrays (FPGAs) are receiving the attention of the Parallel and High Performance Computing community. In this paper, we propose a highly parallel FPGA design for the Floyd-Warshall algorithm to solve the all-pairs shortest-paths problem in a directed graph. Our work is motivated by a computationally intensive bio-informatics application that employs this algorithm. The design we propose makes efficient and maximal utilization of the large amount of resources available on an FPGA to maximize parallelism in the presence of significant data dependences. Experimental results from a working FPGA implementation on the Cray XD1 show a speedup of 22 over execution on the XD1's processor.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have long been used in embedded image and signal processing applications. With rapid advances in modern VLSI technology, FPGAs are becoming increasingly attractive to a much wider audience, in particular, to the High Performance Computing community. Modern FPGAs have abundant resources in the form of tens of thousands of Configurable Logic Blocks (CLBs), a large amount of on-chip memory, and growing numbers of other special-purpose resources. High bandwidth to off-chip memory is sustained through parallel memory access using the large number of I/O pins available on an FPGA. These factors allow FPGAs to exploit a large amount of parallelism and effectively reuse data. Reconfigurability allows very efficient use of available resources tailored to the needs of an application. This makes it possible for custom-designed parallel FPGA implementations to achieve significant speedup over modern general-purpose processors for many long-running routines. This increase in performance has led to the application of FPGAs in HPC systems. Cray and SRC Computers already offer FPGA-based high-performance computer systems [3, 10] that couple general-purpose processors with reconfigurable application accelerators.

In this paper, we leverage the benefits of FPGAs for the Floyd-Warshall (FW) algorithm used to solve the all-pairs shortest-paths problem in a directed graph. The all-pairs shortest-paths problem is to find a shortest path between each pair of vertices in a weighted directed graph. We were particularly motivated with accelerating a long-running bio-informatics code that employs FW [8, 5].

The contributions of this paper are as follows: first, we propose a highly parallel and scalable FPGA design for FW. Second, we build a model to capture the performance of the design and optimize parameters involved. Third, we measure the performance of our working implementation on the Cray XD1, demonstrating a high speedup over a modern general-purpose processor. Experimental results of our design on the Cray XD1 show an improvement by a factor of tens over the CPU implementation. To the best of our knowledge, our work is the first to develop an FPGA-based solution for the all-pairs shortest-paths problem.

The rest of this paper is organized as follows: In Sec-

Supported in part by the Department of Energy's ASC program.

tion 2, we present our motivation behind providing an FPGA-based solution for FW. In Section 3, we give an overview of FPGAs, followed by an overview of the problem. In Section 4, we discuss challenges in extracting parallelism from FW. In Section 5, we propose a new parallel FPGA-based design for FW, extend this design for a tiled algorithm, and discuss how to optimize it to maximize performance. In Section 6, we discuss implementation issues for the design on the Cray XD1. Finally, we present results of our experiments in Section 7.

2 Motivation

The need to analyze data for underlying relationships continues to be a critical element for understanding the behavior of complex systems. This capability is particularly important for analysis of interaction and similarity networks of biological systems. A recent algorithmic development for this type of analysis is the Dynamic Transitive Closure Analysis (DTCA) analysis [15]. Although the method was developed to evaluate undirected graphs representing large gene-drug interaction networks in the study of cancer, it can be used to evaluate any large interaction network. The method incorporates repeated all-pairs shortest-paths evaluations, which are a computational bottleneck for analysis of very large networks.

A scalable implementation of the DTCA algorithm was implemented in a software program called Galaxy as part of the Ohio Biosciences Library [5]. Reading in microarray expression data for several genes and drugs, the program uses the Floyd-Warshall (FW) algorithm to evaluate for closure on multiple subgraphs of the original interaction network. The vertices of the graph represent either genes or drugs under investigation in the study of new therapies for treating cancer. The weight of an edge in the graph is calculated using the co-correlation value computed between each pair of vertices using the microarray expression data provided for each gene and drug involved in the study. The distance used for each edge is $1 - c^2$, where c is the computed co-correlation value. On a genome-wide scale, evaluating graphs exceeding 20,000 nodes is frequently required. The resulting computations required for the multiple $\Theta(n^3)$ FW evaluations result in run-times of several days even for optimized implementations on modern general-purpose processors. A more efficient implementation is sought to reduce evaluation times to more acceptable levels, with a goal of interactive response.

For the application described above, all edge-weights are fractions between 0 and 1, with an accuracy up to three places of decimal desired. Hence, all of these weights can be scaled to integers between 0 and 1000 making a precision of ten bits sufficient. In the context of our problem, all weights are non-negative.

3 Overview

In this section, we give an overview of FPGAs, followed by an overview of the all-pairs shortest-paths problem and the Floyd-Warshall algorithm.

3.1 Modern FPGAs[†]

FPGAs are user-programmable gate arrays with various configurable elements and embedded blocks. At the lowest hardware level, an FPGA is made up of multiple configurable blocks connected to a configurable interconnect, allowing the user to "Field Program" the device for a specific purpose. Current-day FPGAs are optimized for high density and performance. Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. Apart from CLBs, modern FPGAs comprise I/O blocks, Block RAM, embedded dedicated multipliers, and sufficient resources for routing and global clocking.

In the Virtex-II Pro series of FPGAs [14], each CLB includes four slices, and two tri-state buffers that drive dedicated horizontal routing resources. Each slice is equivalent, and comprises two function generators, two storage elements, large multiplexers, arithmetic logic gates, a fast look-ahead carry chain, and a few other resources. Each function generator is configurable as a 4-input look-up table (LUT), as a 16-bit shift register, or as 16-bit distributed RAM. Each CLB has an internal fast interconnect and connects to a switch matrix to access general routing resources. Apart from CLBs, the Virtex family provides large quantities of dual-ported RAM blocks, allowing the user to store more data on-chip, and create deeper and wider pipelines. Each block RAM resource is an 18Kb dual-port RAM, programmable in various depth and width configurations. Cray has chosen several members of the Xilinx Virtex-II Pro and Virtex-4 families for use in the XD1 (Table 1).

3.2 Cray XD1

A Cray XD1 system is composed of multiple chassis, each containing up to six compute blades. Each compute blade (Fig. 1) contains two single- or dual-core 64-bit AMD Opteron processors, 1 to 8 GB of DDR-SDRAM per processor, a RapidArray processor which provides two 2 GB/s RapidArray links to the switch fabric, and an application acceleration module [3]. The application acceleration module is an FPGA-based reconfigurable computing module that provides an FPGA complemented with a RapidArray Transport core providing a programmable clock source, and four

[†]This overview applies to most modern FPGAs, but many details are specific to the Xilinx Virtex-II Pro series.

FPGA	Max clock rate	Slices	Block RAM (Kb)	I/O pins
XC2VP50	200 MHz	23,616	4,176	852
XC2VP100	200 MHz	44,096	7,992	1164
XC4VLX160	500 MHz	67,584	5,184	960

Table 1. Some available Xilinx Virtex-II Pro and Virtex-4 FPGAs.

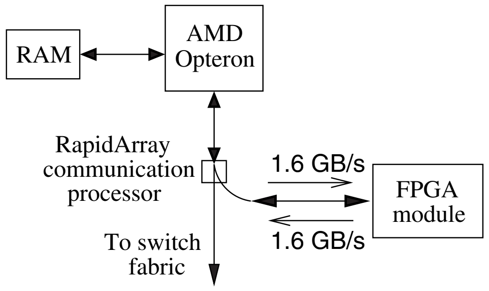


Figure 1. The Cray XD1 system

banks of Quad Data Rate (QDR) II SRAM. Cray presently offers a choice between 8, 16, or 32 MB of SRAM.

3.3 The All-Pairs Shortest-Paths problem

Given a weighted, directed graph $G = (V, E)$ with a weight function, $w : E \rightarrow \mathbf{R}$, that maps edges to real-valued weights, we wish to find, for every pair of vertices u, $v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. Output is typically desired in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v.

3.4 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm uses a dynamic programming approach to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$ [2, 6, 13]. It runs in $\Theta(|V|^3)$ time. Let

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{weight of edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E. \end{cases}$$

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. For $k = 0$, we have $d_{ij}^{0} = w_{ij}$. A recursive definition from the above formulation is given by:

$$d_{ij}^{k} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right) & \text{if } k \geq 1 \end{cases}$$

```
Input: $d[1 \ldots N, 1 \ldots N]$
1: for $k \leftarrow 1, N$ do
2:     for $i \leftarrow 1, N$ do
3:         for $j \leftarrow 1, N$ do
4:             $d[i, j] \leftarrow \min(d[i, j], d[i, k] + d[k, j])$
5:         end for
6:     end for
7: end for
Output: $d[1 \ldots N, 1 \ldots N]$
```

Figure 2. The Floyd-Warshall algorithm

The matrix $\{d_{ij}^{N}\}$, $1 \leq i, j \leq N$, gives the final result. The above recursive definition can be written as a bottom-up procedure shown in Fig. 2. We refer to the matrix d as the distance matrix in the rest of this paper.

The FW code is tight with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Unlike many graph algorithms, the absence of the need to implement any complex abstract data types makes FW a good candidate for acceleration with an FPGA.

4 Design challenges

In this section, we discuss the challenges and issues in implementing FW on FPGAs.

4.1 Parallelism and data reuse

In the FW computation, we have exactly N^2 data elements, but $\Theta(N^3)$ computations to perform. Hence, there is high temporal locality that a custom design can exploit. Parallelism on an FPGA is often achieved through employing a large number of processing elements working together in a particular fashion. Locality is exploited by storing data in the block RAM on the chip, from which it can be read or written to in a single clock cycle. The block RAM does not count as part of the slices that the logic of the design occupies.

4.2 Reconfiguration cost

The cost of reprogramming an FPGA may be prohibitively high so as to not allow using different optimized kernels for different cases that may arise for a problem. Runtime reconfiguration would be particularly useful when the cost of reconfiguration can be offset by the benefit that can be obtained by using specific compute kernels for specific tasks. On the XD1 FPGA, the cost of reprogramming is currently about 1.6 s. Hence, a feasible design for FW would be a single kernel that when loaded on the FPGA would be capable of performing FW for all matrix sizes.

4.3 Data dependences

The FW nested code looks similar to matrix multiplication, but has additional Read-after-Write and Write-after-Read dependences in addition to the output dependences that exist in matrix multiplication. The problem of extracting a large amount of parallelism in the presence of these dependences is not trivial, and in fact, doing so making maximum use of available FPGA resources is a major challenge. At the same time, the design needs to be simple and modular in order to achieve a high target clock rate. Also, the design should scale well when – (1) larger FPGAs are employed, and (2) higher I/O bandwidth to the system is available.

5 Parallel FPGA design for FW

Let d be the distance matrix of a directed graph of B nodes. Let us now more closely observe the computation being performed in the nested loop of Fig. 2.

At any iteration $k = r$ of the outermost loop, the vectors, $d[r,*]$ and $d[*,r]$, update the whole matrix d. Let us call this row and column, the *pivot row* and the *pivot column*, respectively. Note that $d[r,*]$ and $d[*,r]$ are not updated during iteration $k = r$, as $d[r,r] = 0$. Let p_1^r and p_2^r denote the pivot row and the pivot column, respectively, of iteration $k = r$. From the notation introduced in Sec. 3.4, $d^r[r,*]$ and $d^r[*,r]$ are the r^{th} row and r^{th} column of the matrix after r iterations of the k loop. Hence, we have:

$$p_1^r[i] \quad = \quad d^r[r,i], \quad 1 \le i \le B \qquad (1)$$

$$p_2^r[i] \quad = \quad d^r[i,r], \quad 1 \le i \le B \qquad (2)$$

5.1 A naive approach

If the k loop is processed sequentially, parallelism can be extracted for a given k only by reading in and updating several elements of a row/column simultaneously. Using this naive approach, the degree of parallelism would be same as the number of matrix elements that can be read in and updated simultaneously. For a given iteration of the k loop, all elements of the matrix are updated exactly once. This approach would require the matrix (or the tile being processed) to be stored in the block RAM of the FPGA. If a batch of row (or column) elements of the tile need to be processed in parallel, they would require a single element of the pivot column (or row). This would create a fanout problem if a large number of operators work in parallel. Second, there is significant complexity involved in distributing the tile across BRAMs and in addressing it. Third, the amount of parallelism that can be extracted using this approach cannot be increased beyond a certain limit no matter how large the FPGA is. All of these factors severely effect the degree of parallelism and scalability of this design apart from making it infeasible to build.

5.2 New parallel architecture

Central Idea. We propose a new approach that extracts parallelism even from the k loop in the presence of dependences. This is achieved by computing in advance, the pivot rows and columns that would update the array for any iteration of the outermost loop k. The r^{th} pre-computed pivot row-column pair is stored in the r^{th} processing element (PE) in a linear array of PEs. Pipelined parallelism from the array of PEs is used for both – initially computing the pivot rows and columns, and subsequently updating the matrix with the pre-computed pivot rows and columns.

The r^{th} PE is made responsible for updating the matrix elements with the pivot row and column it has in its storage. The work done by the r^{th} PE across time corresponds to the computation in the iteration $k = r$ of the outermost loop of FW. The above two-phased approach is described in detail below.

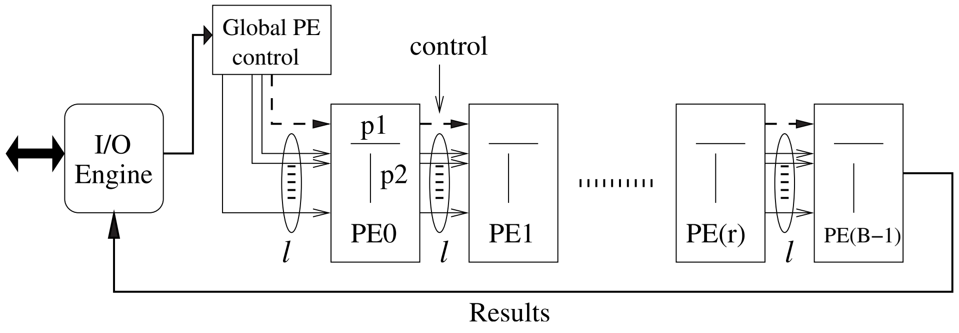
Let B be the number of PEs. Let an operator comprise a comparator and an adder of required width. Let l be the number of operators in each PE. So, l elements can be read in parallel by PE$\langle i \rangle$, updated and shifted to PE$\langle i+1 \rangle$. In the rest of this section, by an "update", we mean the computation shown in Fig. 3(b); by "upstream" and "downstream" with respect to a particular PE, we mean the PEs that are to its left and right, respectively, in the linear array of PEs. B and l are determined by resource constraints as described later.

Phase 1: At each clock cycle, l elements of row i (or column i, alternating between the two) of a BxB matrix in the system's memory are streamed in to PE$\langle 0 \rangle$, i going from 0 to $B-1$, i.e., row 0 followed by column 0, row 1 followed by column 1, and so on. These elements are updated until they reach PE$\langle i \rangle$, at which they get stored. The idea involved is that the r^{th} row (or column) gets updated $r-1$ times (as it is shifted through the PEs) before getting stored at PE$\langle r \rangle$ as that PE's pivot row and column. This process is done in a pipelined fashion.

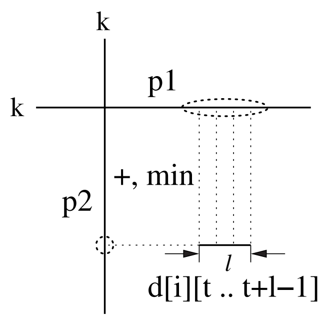
This computation of pivot rows and columns involves the same set of operations as in the original FW code, with the dependences preserved. Hence, p_1^r and p_2^r are in the storage of PE$\langle r \rangle$ after it finishes this phase.

Time spent by PE$\langle r \rangle$ in phase 1 is given by:

$$T_1(r) = \frac{2B*(B-r)}{l} \text{ clock cycles} \qquad (3)$$



(a) FW kernel architecture



(b) Parallel update at PE$\langle k \rangle$

Figure 3. Parallel FPGA-based FW kernel

Phase 2: Since the pivot row and column elements in the local storage of each PE are themselves partially computed distances, we need not stream in the matrix again. Instead each PE, in its turn, starting from PE$\langle 0 \rangle$, streams its pivot row (or column) downstream to be computed on and updated. Hence, PE$\langle r \rangle$ would update and shift r rows of the matrix it would receive from the previous PE. Then on its turn, it streams its row which is updated and shifted $B-r$ times by the downstream PEs. The last PE would output all B^2 elements of the result in row-major order (or column-major if columns were streamed out). Fig. 3 shows the architecture of the design. Time spent by PE$\langle r \rangle$ in phase 2 is given by:

$$T_2(r) = \frac{B*(r+1)}{l} \text{ cycles} \tag{4}$$

Summary of work done by each PE. Let PE$\langle r \rangle$ start at $t=0$. For the first $2B/l$ clock cycles, it would receive its pivot row and pivot column, each with B elements. For the next $2(B/l)(B-r-1)$ clock cycles, it would update batches of elements coming from its left neighbor which are actually meant to be the pivot row and column elements for PEs downstream; on finishing this, the PE enters the second phase. In the second phase, for the first $B*r/l$ clock cycles, it would update and shift elements read from the previous PE, which are actually the pivot row elements coming from the PEs upstream. It would then finish up by sending its own pivot row elements in B/l clock cycles. The pipeline empties from left to right as each PE finishes with sending its pivot row elements.

Let p be the pipeline depth of a PE. Then, the latency of the design is given by:

$$\begin{aligned} L &= T_1(0) + B*p - 1 + T_2(B-1) \text{ cycles} \\ &= \left(\frac{3B^2}{l} + B*p - 1\right) \text{ cycles} \end{aligned} \tag{5}$$

5.3 Extending the design for a tiled FW algorithm

The problem of tiling the Floyd-Warshall algorithm in order to optimize it for the cache hierarchy on general-purpose processors has been addressed by Venkataraman et al. [12]. Since the FW kernel described above can handle only matrices of size BxB, we extend the above design to handle the tiling scheme proposed in [12]. We first give a brief description of the same.

Consider the operation shown in Fig. 3(b). Replace each element by a tile of size BxB so that there are $(N/B)^2$ tiles, and consider a similar operation happening to tiles. In this case, we have an entire tile that needs to be updated by the projection of its elements onto pivot rows and columns that come from a pivot row-block ($d[t\ldots t+B-1][1\ldots N]$) and a pivot column block ($d[1\ldots N][t\ldots t+B-1]$) for the outermost loop $k=t$ to $t+B-1$. The pivot rows and columns used to update a particular tile may – (1) come from the same tile (self-dependent), (2) only the pivot rows come from a different tile (partially row-dependent), (3) only the pivot columns come from a different tile (partially column-dependent), or (4) both the pivot rows and the pivot columns come from different tiles (doubly-dependent). Fig. 4 shows these different types of tiles.

The partially row/column-dependent tiles require the self-dependent tile to be processed first. Similarly, the doubly-dependent tiles depend on the row-dependent and column-dependent tiles (Fig. 4). For any large $N > B$, we choose a tile size of B. For cases (2), (3) and (4), in which one or both of the pivot rows and columns come from tiles different from the one we are updating, no updates need to be performed on the pivot elements as they are shifted in Phase 1. In the second phase of the algorithm, for the doubly-dependent case, the tile that is to be updated needs to be streamed in as opposed to PEs themselves sending their pivot row elements downstream. It is to be noted that for case (2), the PEs send their pivot column elements downstream as opposed to pivot row elements. Hence, minor changes to the control instructions sent to the PEs allow

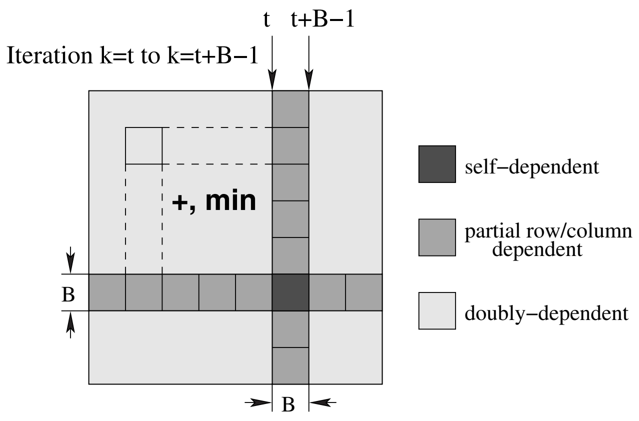


Figure 4. Tiled FW algorithm proposed in [12]

us to accomplish this.

The latency for all types of tiles is the same as that for self-dependent tiles, and is given by Eqn. 5.

5.4 Parallelism (B and l)

The factors B and l represent two orthogonal ways in which parallelism is extracted in our design. l is governed by the I/O bandwidth and the size of the matrix values. B is constrained by the amount of FPGA resources – the number of slices or the amount of block RAM. The product of B and l is the degree of parallelism of our design, and maximizing this is our goal.

In the presence of lower I/O bandwidth, l can be decreased which may free resources to increase B. If higher I/O bandwidth is available, l can be increased leading to more parallelism. Likewise, larger number of slices on an FPGA would allow us to have a higher value for B. Hence,

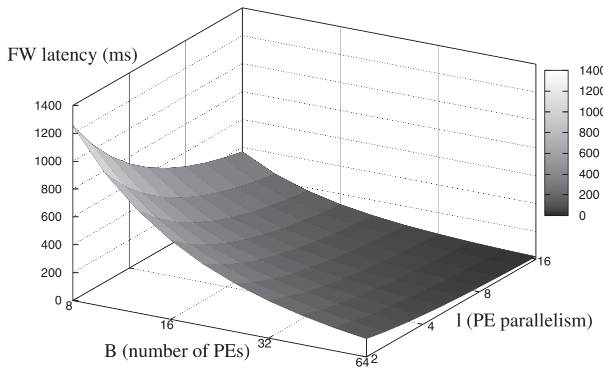


Figure 5. Effect of parameters B and l on the latency of FPGA-based FW

our design always makes maximum utilization of FPGA resources even as larger quantities of it are available. For the purpose of illustrating the effect of parameters, we obtain the FW latency for an NxN matrix by simply multiplying $(N/B)^3$, which is the number of tiled computations, with the per-tile latency from Eqn. 5. Fig. 5 shows this latency for a 1024x1024 matrix with 16-bit edge-weights. However, this would not be a true measure of the implementation's application-level performance as discussed later.

5.5 Determining optimal values for B and l

We use the following notation throughout the rest of this paper:

M:	Amount of available block RAM
S :	Number of slices available for FW kernel
c_g:	Number of slices occupied by global control
c_p:	Number of slices occupied by each PE's control
s_o :	Number of slices occupied by an operator
t_c :	Target clock cycle time achievable
d :	Size of an edge-weight (in bytes)
w :	Number of bytes that can be fetched per cycle
p :	Depth of the PE pipeline

Storage constraint: Each PE stores $2B$ elements of matrix d. For all PEs to have sufficient storage for their pivot rows and columns:

$$2B^2 d \leq M \tag{6}$$

Area constraint: The number of slices utilized by the design should be within the number available on the FPGA.

$$s_o B l + c_p B + c_g \leq S \tag{7}$$

From the above two constraints, we have:

$$B \leq \min\left(\frac{S - c_g}{s_o l + c_p}, \sqrt{\frac{M}{2d}}\right) \tag{8}$$

I/O bandwidth constraint: With l operators working in parallel in a single PE, we need to have: $l * d \leq w$.

An increase in l leads to an increase only in the number of operators, while increasing B would add both, operators and control logic. Thus, the parameter l can be thought of as providing more parallelism per unit area than B. Hence, to maximize parallelism which is the product of B and l, we set l to the maximum value possible, i.e., w/d, and then determine B from the area constraint. The optimal values l^* and B^* are given by:

$$l^* = \frac{w}{d}; \qquad B^* = \min\left(\frac{S - c_g}{s_o l^* + c_p}, \sqrt{\frac{M}{2d}}\right) \tag{9}$$

To be precise, the optimal values of B and l are the highest powers of two less than the corresponding values on the right-hand side of the above equation.

Since $s_o \ll c_p$ (Table 2), to increase B, it is important to reduce c_p, and transfer as much control as possible to global control, c_g. This is dealt with in Sec. 5.6.

5.6 Control design and optimization

Each PE takes only one of the following actions at any clock cycle:

1. **Read and Store**: Read and store a row (or column) batch (set of l elements) in its local storage
2. **Process**: Read, update and shift a row (or column) batch
3. **Forward**: Read and shift a row (or column) batch
4. **Send**: Send a batch from a stored pivot row (or column) to the next PE

Control instructions can be streamed through the array of PEs eliminating the need for each PE to maintain a finite state machine based on the number of elements received or the number of clock cycles. Each batch of l elements has a control instruction associated with it. The improvement due to this optimization is significant as the amount of control logic eliminated in each PE is comparable to the slices occupied by all the l operators in it. For the sake of brevity, we do not discuss the instruction format and the corresponding action taken by a PE, but by having an index of $\log_2 B$ bits and a three-bit opcode, it is possible for a PE to infer the action it has to perform, and the address in the pivot row and column RAM to read from, if necessary. A finite state machine need only be maintained at the global PE control shown in Fig. 3(a).

5.7 Overlapping successive tile computations

Computation for successive tiles can be overlapped, i.e., emptying and filling of the PE pipeline can be overlapped. In this case, the latency for doubly-dependent tiles is different from that of self-dependent and partial row/column-dependent tiles. In the former case, the first PE can start processing the next tile after $3B^2/l$ clock cycles, while in the latter, the next tile can be processed $(2B^2/l + B/l)$ cycles after the first tile is sent in.

Let L_{sd}, L_{rd}, L_{cd} and L_{dd} be the latencies of self-dependent, partially row-dependent, partially column-dependent and doubly-dependent tiles, respectively. These latencies are given by:

$$L_{sd} = L_{rd} = L_{cd} = \left(\frac{2B^2}{l} + \frac{B}{l}\right) t_c \qquad (10)$$

$$L_{dd} = \left(\frac{3B^2}{l}\right) t_c \qquad (11)$$

5.8 Optimizing non-self-dependent tile computations

Latency for processing partially or doubly-dependent tiles can be reduced since a set of tiles in a particular row (or column) block of the array use the same set of pivot columns (or rows). The pivot columns (or rows) for all tiles in the row (or column) block need only be fetched once per that row (column) block. Therefore, the latencies in Eqn. 10 and Eqn. 11 reduce by an amount B^2/l when appropriate.

5.9 Pipelining for a high frequency

To achieve a high target clock rate, we use a three-stage pipeline for the PE. In the first stage, the control instruction is decoded and addresses for the PE's pivot row and column are placed. In the second stage, the pivot row/column elements are read. The third stage performs the add, compare and update operation.

The purpose of introducing the second and the third stage is to separate reading of the pivot row/column elements from the block RAM, and the add/compare/update operation. Completing both of them in a single clock cycle led to a clock rate significantly less than 200 MHz, the maximum that can be achieved from the XC2VP50 (Table 1). Also, headroom needs to be left in case higher precision is desired. Pipelining this way allows us to clock the design at a full 200 MHz. We do not analyze the trade-off between pipelining and frequency as the benefit obtained by the higher clock rate was definitely desirable due to the amount of block RAM never becoming a constraint (Eqn. 8).

6 Implementation

In this section, we briefly describe issues in implementing the proposed parallel FW design on the Cray XD1.

6.1 Multi-level tiling

FPGAs have many times higher bandwidth to off-chip SRAM than over the interconnect to the system. Hence, an optimized design on reconfigurable HPC systems like the Cray XD1, would make use of SRAM as an additional hierarchy in tiling. We would have two levels of tiling – one with respect to the maximum tile size which the FPGA can handle due to its resource constraints, and the second with respect to the SRAM. The SRAM being several megabytes in size can accommodate a much larger tile than the FPGA can handle. A design tiled with respect to the SRAM would reduce I/O between the FPGA and the system by a factor proportional to the ratio of the SRAM tile size to the tile size the FPGA kernel can handle.

Though tiling with respect to the SRAM is conceptually straightforward, it is tedious to implement with the current state of tools. Hence, in the design for which results are presented in the next section, we do not tile with respect to the SRAM.

6.2 I/O bandwidth and precision

Though our design is parameterized with respect to the precision of the distance matrix values, our measurements in the following section are for 16-bit values.

On the Cray XD1, at the theoretical peak rate of 1.6 GB/s, a 64-bit word can be read and written every clock cycle over the interconnect. Hence, four matrix elements can be read in a single clock cycle. However, the theoretical peak bandwidth over the interconnect is not achieved due to burst requirements and DMA overhead. The design takes care of this by stalling the PEs in a cascading fashion (leading to bubbles in the pipeline) whenever data is not available at the peak rate.

Pivot row/column storage. On-chip Block RAM is utilized for the storage of a pivot row and column in each PE. The width of each pivot RAM word is $l * d$, with B/l such words. Block RAM on the Virtex-II Pro FPGAs can be configured for a desired depth and width, starting from 18Kx1-bit to 36x512-bit. This makes sure that we are not constrained when reading a large number of pivot row/column elements when higher I/O bandwidth is available.

6.3 FPGA-host communication

The FPGA can perform I/O only to a specially allocated buffer that is pinned to the system's memory. The physical address of this transfer region is communicated to the FPGA.

We keep the I/O engine on the FPGA simple. The I/O engine transfers data of specified length to/from contiguous regions of the communication buffer, and is totally oblivious to the computation being performed by the FW kernel. Rows and columns of the distance matrix are placed in the communication buffer in the fashion required by the design, for the kind of tile to be processed. $3B^2$ elements are placed in the communication buffer for doubly-dependent tiles, while $2B^2$ elements are placed for the other three cases. A special set of registers on the FPGA are used for transfer of control and status information from and to the application running on the system. The source and destination buffer addresses, amount of data to be transferred to/from these buffers, and the type of tile to be processed, is communicated using these registers.

The Cray User-FPGA API provides functions to program the FPGA, create the communication buffer, write values and addresses to registers on the FPGA, taking care of virtual to physical address translation in the latter case. Resetting the logic on the FPGA is done through a write to a special register. A write to the register meant for the destination buffer address triggers the computation on the FPGA. Completion is indicated by setting a bit which the CPU polls.

7 Measurements

In this section, we measure the performance of FPGA-based FW, and compare it that of FW on XD1's CPU.

7.1 Measurements on a modern microprocessor

The measurements for the general-purpose processor case were taken on a 2.2 GHz 64-bit AMD Opteron with a 64 KB L1 data cache and a 1 MB L2 cache as found on the XD1. GCC 3.3 with "-O3" was used for compilation. The latency reported is averaged over 1000 iterations. Since the dataset for all the cases shown in Table 4 fits in the L1 data cache, we do not perform any tiling, or copying to a contiguous buffer to reduce conflict misses. We refer to the system implementation as CPU-FW in the rest of this section.

7.2 Measurements for FPGA-based FW

The FPGA on the Cray XD1 is a Xilinx Virtex-II Pro XC2VP50 (Table 1). Xilinx ISE 7.1i was used for synthesizing, mapping, placing, and routing the design. The version of Cray User-FPGA API used was 1.2. The Operating System on the XD1 is Linux kernel version 2.6.5. The FPGA-based FW implementation is referred to as FPGA-FW in the rest of this section.

Resource utilization. Table 2 gives the resource utilization of different modules of FPGA-FW as reported by the mapping tool. Since we have 16-bit distance values, $d = 2$. Since four such values can be fetched in a clock cycle, we have $l = 4$. The PE pipeline depth p is three as mentioned in Sec. 5.9. We clock FPGA-FW at the maximum of 200 MHz, i.e., $t_c = 5$ ns. The XC2VP50 has over 200 18Kb dual-ported Block RAMs. As the block RAM is also used for other purposes – mainly for the interstage buffers of the PEs' pipelines, we fix M at a conservative 128 KB. Substituting these values into Eqn. 9, we find that the highest power of two that B can assume for the XC2VP50 FPGA is 32. In Table 3, we give an estimate of the tile sizes

Area group	Number of Slices		
	8x8	16x16	32x32
Operator (s_o)	25	25	25
PE	584	550	553
Global control (c_g)	73	73	73
FW	3,983	8,256	17,223
I/O subsystem	3,193		
Total utilization	8,017	12,293	21,229
Block RAM utilization	48	80	144
Available (A)	23,616		

Table 2. Resource utilization for FPGA-FW on the Xilinx XC2VP50 FPGA

FPGA	Available Slices	B	
		l = 4	l = 16
XC2VP50	23,616	**32**	16
XC2VP100	44,096	64	32
XC4VLX160	67,584	64	64

Table 3. Tile size supported by the largest FPGA-FW kernel that can fit on various FPGAs

that would be possible on some larger FPGAs.

Per-tile overhead. The latency of processing a single tile includes certain overhead that does not show up when the tiled implementation is used iteratively for a full-fledged matrix. We make an effort to characterize this overhead. This overhead can be measured by setting the number of bytes to be transferred to the destination buffer to zero: in this case, as soon as the FW kernel receives its first batch of input data, completion is indicated. Hence, we get the interval of time between the point when a call to reset is made in the user application and the point when the FW kernel starts computation. We subtract this overhead from the total time measured: the result is the per-tile latency that is an indicator of performance of the FW kernel. We find the overhead to be a constant 2.1 μs.

To time FPGA-FW, we use the measured latency averaged over 1000 iterations. Each iteration is a write to the reset register followed by a destination address write that triggers the computation, that finishes by polling of the completion flag by the CPU. Table 4 gives a comparison of the measured performance of FPGA-FW and CPU-FW.

Table 5 compares the latency estimated from Eqn. 5 and the one actually measured, and shows the speedup achieved with the FPGA over the microprocessor. It is to be observed that the measured latencies for various tile sizes closely agree with the estimated values. The deviation from the estimated value is due to the fact that the available I/O bandwidth is less than the theoretical peak. Fig. 6 gives a breakdown of the latency for processing a tile.

Tile size	FPGA-FW			CPU-FW
	Total	Overhead	Compute	
8x8	2.49μs	2.07μs	0.42μs	1.6μs
16x16	3.36μs	2.07μs	1.29μs	14.1μs
32x32	6.91μs	2.07μs	4.84μs	106.5μs

Table 4. Measured performance comparison of FW: FPGA-FW vs. CPU-FW

Tile Size	FPGA-FW		CPU-FW	Measured Speedup
	Estimated	Measured		
8x8	0.36μs	0.42μs	1.6μs	3.8x
16x16	1.20μs	1.29μs	14.1μs	11x
32x32	4.14μs	4.84μs	106.5μs	22x

Table 5. FPGA-FW: Estimated vs. measured performance and speedup over CPU-FW

A high speedup is obtained without making use of off-chip SRAM. Apart from the large degree of parallelism that is extracted, adapting the FPGA resources for the appropriate amount of precision necessary for the problem at hand, is an important factor responsible for the acceleration. For example, for the 32x32 case, we have 128 (= 32x4) operators working in parallel each clock cycle. All of these benefits heavily offset the downside of having a clock rate on the FPGA that is almost ten times lower than that of the microprocessor.

8 Related work

The Floyd-Warshall algorithm was first proposed by Robert Floyd [6]. Floyd based his algorithm on a theorem

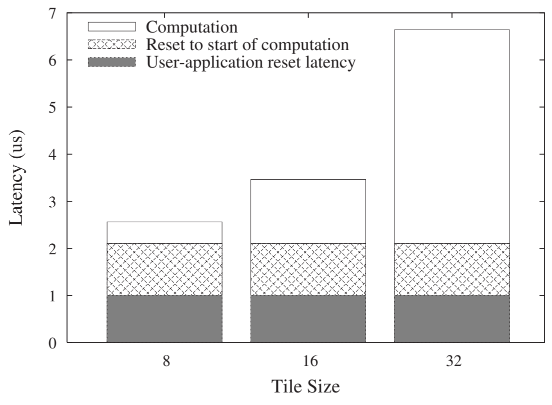


Figure 6. Breakup of FPGA-FW latency

of Warshall [13] that describes how to compute the transitive closure of boolean matrices. Venkataraman et al. [12] proposed a blocked algorithm to optimize it for the cache hierarchy of modern processors.

Researchers have recently demonstrated the competitiveness of FPGAs with modern processors for double-precision floating-point arithmetic and dense linear algebra operations [11, 16]. A significant amount of work also proposes FPGA designs for specific applications demanding high performance. However, most of these studies do not provide experimental data, but only project performance.

A significant amount of systolic literature exists on the transitive closure problem and its generalized form - the Algebraic Path Problem [7, 9, 4]. However, designs proposed in these works do not address practical resource constraints that necessitate a tiled solution for a large problem size.

9 Future work

A number of design trade-offs and implementation choices arise when a large *N*x*N* matrix is to be processed using the FPGA kernel we have developed. Some of the issues include – layout transformation for the distance matrix, reducing and hiding copy costs to the FPGA communication buffer, and eliminating or hiding the start-up overhead of the FW kernel. The intent of this work is to provide a high-performance FPGA kernel to process a tile efficiently; this kernel can be used to build a complete tiled solution for a graph with a large number of vertices [1]. We plan to integrate the final implementation into Galaxy, and reduce the running time of the application significantly.

10 Conclusions

In this paper, we have proposed a parallel FPGA design for the Floyd-Warshall algorithm to solve the all-pairs shortest-paths problem in a directed graph. In order to utilize parallelism without data access conflicts, the computation was reorganized into a sequence of two passes: first compute a set of *pivot* rows and columns, and then use the stored pivot rows and columns to compute the updates to matrix elements in a streamed fashion. This approach enabled the creation of a simple and modular design that maximizes parallelism and makes maximal use of the resources available on the FPGA. A model was constructed to determine the optimal values of parameters that govern the exploitable degree of parallelism in the design under resource constraints. The implemented kernel can be used to develop a tiled algorithm [12] for highly accelerated solution of large all-pairs shortest-paths problems. Experimental results from a working implementation of the kernel show a speedup of 22 on the Cray XD1.

References

[1] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/Software Codesign for All-Pairs Shortest Paths on a Reconfigurable Supercomputer. Technical Report OSU-CISRC-1/06-TR13, Jan. 2006.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[3] Cray Inc. Cray XD1 whitepaper, 2005.

[4] C. T. Djamégni, P. Quinton, S. V. Rajopadhye, and T. Risset. Derivation of Systolic Algorithms for the Algebraic Path Problem by Recurrence Transformations. *Parallel Computing*, 26(11):1429–1445, Oct. 2000.

[5] D. P. Dougherty, E. A. Stahlberg, and W. Sadee. Network Analysis Using Transitive Closure: New Methods for Exploring Networks. *Journal of Statistical Computation and Simulation*, 2005.

[6] R. W. Floyd. Algorithm 97: Shortest Path. In *Communications of the ACM*, volume 5, page 345, June 1962.

[7] S.-Y. Kung, S.-C. Lo, and P. S. Lewis. Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems. *IEEE Transactions on Computers*, 36(5):603–614, May 1987.

[8] Ohio Supercomputer Center. Galaxy. http://www.osc.edu/hpc/software/apps/galaxy.shtml.

[9] D. Sarkar and A. Mukherjee. Design of Optimal Systolic Algorithms for the Transitive Closure Problem. *IEEE Transactions on Computers*, 41(4):508–512, Apr. 1992.

[10] SRC Computers Inc. SRC MAPstation.

[11] K. D. Underwood and K. S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, Apr. 2004.

[12] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A Blocked All-Pairs Shortest-Paths Algorithm. *Journal of Experimental Algorithmics*, 8:2.2, Dec. 2003.

[13] S. Warshall. A Theorem on Boolean Matrices. In *Journal of the ACM*, volume 9, pages 11–12, Jan. 1962.

[14] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. Xilinx Inc., 2005.

[15] X. Zhou, M. Kao, and W. Wong. Transitive Functional Annotation by Shortest-Path Analysis of Gene Expression Data. In *P. Natl. Acad. Sci., USA*, 2002.

[16] L. Zhuo and V. K. Prasanna. Design Trade-offs for BLAS Operations on Reconfigurable Hardware. In *Proceedings of the International Conference on Parallel Processing (ICPP'05)*, pages 78–86, June 2005.