# Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths

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

[1]*Department of Computer Science & Engineering*
*The Ohio State University*
*Columbus, OH 43210*
{*bondhugu, dinan, 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, eas*}*@osc.edu*

## Abstract

*Field-Programmable Gate Arrays (FPGAs) are being employed in high performance computing systems owing to their potential to accelerate a wide variety of long-running routines. Parallel FPGA-based designs often yield a very high speedup. Applications using these designs on reconfigurable supercomputers involve software on the system managing computation on the FPGA. To extract maximum performance from an FPGA design at the application level, it becomes necessary to minimize associated data movement costs on the system. We address this hardware/software integration challenge in the context of the All-Pairs Shortest-Paths (APSP) problem in a directed graph. We employ a parallel FPGA-based design using a blocked algorithm to solve large instances of APSP. With appropriate design choices and optimizations, experimental results on the Cray XD1 show that the FPGA-based implementation sustains an application-level speedup of 15 over an optimized CPU-based implementation.*

## 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 (HPC) 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-chip. All of these factors allow FPGAs to extract a large amount of parallelism apart from effectively reusing 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 use of FPGAs in HPC systems. Cray and SRC Computers already offer FPGA-based high-performance computer systems that couple general-purpose processors with reconfigurable application accelerators [3, 9].

A complete FPGA-based solution on reconfigurable supercomputers like the Cray XD1 involves an FPGA design integrated with a user-application running on the system. Limited resources on the FPGA often necessitate a blocked algorithm for the problem. This is particularly true for many linear algebra routines [16, 17]. Large instances of a problem are solved by employing a parallel FPGA design iteratively with a blocked al-

This research is supported in part by the Department of Energy's ASC program.

gorithm. Parallel FPGA designs often yield significant speedup. Therefore, in many cases, it becomes important to orchestrate movement of subsets of data (tiles) so that benefits are reflected at the application level. For some scientific routines involving double-precision floating-point arithmetic, performance of current day FPGAs may not be high enough to make system-side optimizations crucial. However, with current trends in double-precision floating-point performance of FPGAs [11, 12], FPGA performance would eventually reach a level to make these optimizations beneficial for all FPGA-based routines.

The all-pairs shortest-paths problem is to find a shortest path between each pair of vertices in a weighted directed graph. The Floyd-Warshall (FW) algorithm used to solve this problem involves nested loop code that exhibits a regular access pattern with significant data dependences. In the context of FW, we propose approaches that solve the following problem: given an FPGA kernel that achieves a high speedup on a small dataset (tile/block), what optimizations at the system would help obtain a high percentage of this speedup at the application level for large problem sizes? The parallel kernel we use is from our previous work [1]. With appropriate design choices and optimizations, we show through experimental results on the Cray XD1 that the application-level speedup for FPGA-based FW improves from 4 to 15. We build a model that accurately captures performance of the FPGA-based implementation, and provides insights into factors affecting it.

The rest of this paper is organized as follows: In Section 3, we give an overview of the Cray XD1 followed by an overview of the FW algorithm, and the FPGA-based FW design developed in [1]. In sections 4 and 5, we discuss design choices and propose techniques to reduce data movement costs on the system. In Section 6, we present results of our experiments on the Cray XD1 and analyze performance in detail.

## 2. Related work

The Floyd-Warshall algorithm was first proposed by Robert Floyd [5]. Floyd based his algorithm on a theorem of Warshall [14] that describes how to compute the transitive closure of boolean matrices. Venkataraman et al. proposed a blocked implementation of the algorithm to optimize it for the cache hierarchy of modern processors [13].

Apart from the Floyd-Warshall algorithm, all-pairs shortest-paths (APSP) algorithms with lower time complexity exist. Karger [6] solved undirected APSP with non-negative edge-weights in $O(Mn + n^2 \log n)$ time, where $n$ is the number of vertices in the graph and $M$ is the the number of edges participating in shortest-paths. Zwick [8] obtained an $O(n^{2.575})$ APSP algorithm where the dependence of the running time is polynomial in the maximum magnitude of the edge weights. It is thus only effective when the edge weights are integers of small absolute value.

Researchers have recently demonstrated the competitiveness of FPGAs with modern microprocessors for double-precision floating-point arithmetic and linear algebra routines [11, 12, 16, 17]. All of these works propose parallel designs and project performance. Tripp et al. [10] study aspects of hardware/software integration on reconfigurable supercomputers with a traffic simulation FPGA kernel. Our work addresses optimizing data movement, and other issues that often arise in mapping nested loops.

In [1], we proposed a parallel FPGA-based design for FW to process a tile efficiently; the kernel is suitable for accelerated solution of large APSP problem instances using a blocked algorithm [13].

## 3. Overview

In this section, we give an overview of the Cray XD1 system, the Floyd-Warshall (FW) algorithm, the parallel FPGA-based FW kernel, and the blocked FW algorithm.

### 3.1 Cray XD1

The Cray XD1 system is composed of multiple chassis, each containing up to six compute blades. Each compute blade contains two single- or dual-core 64-bit AMD Opteron processors, 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 (RT) core providing a programmable clock source, and four banks of Quad Data Rate (QDR) II SRAM.

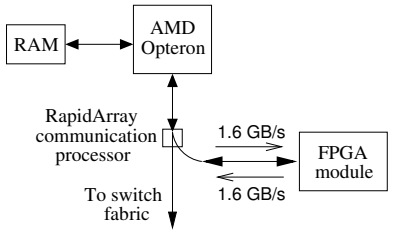


**Figure 1. The Cray XD1 system**

### 3.2 The Floyd-Warshall algorithm

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 $v$.

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**
8: **Output:** $d$

**Figure 2. 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 [2, 5, 14]. It runs in $\Theta(|V|^3)$ time.

Let $w_{ij}$ be the weight of edge $(i,j)$. $w_{ij}$ is 0 when $i = j$, and is $\infty$ when $(i,j) \notin E$. 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:

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

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 as shown in Fig. 2. The code is tight with no elaborate data structures, and so the constant hidden in the $\Theta$-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.

### 3.3 Parallel FW design for a blocked algorithm

In this section, we give a brief description of the parallel FPGA-based FW kernel that we designed in [1].

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. The FW nested loop code has significant data dependences. Extracting parallelism in the presence of these dependences without data access conflicts making maximum use of available FPGA resources is a major challenge.

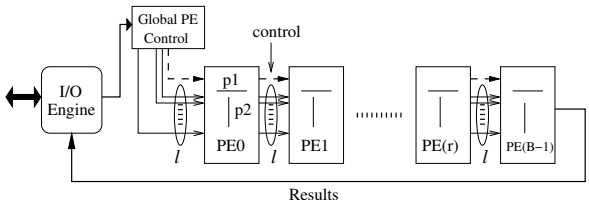


**Figure 3. Parallel FW kernel architecture**

Let $d$ be the distance matrix of a directed graph of $B$ nodes. In the nested loop code shown in Fig. 2, at any iteration $k = r$ of the outer loop, the vectors, $d[r,*]$ and $d[*,r]$, update the whole matrix $d$. We call this row and column, the *pivot row* and the *pivot column*, respectively. In order to extract parallelism from the outermost $k$ loop, the computation was reorganized into a sequence of two passes: in the first pass, compute the set of *pivot* rows and columns, and then use the stored pivot rows/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. The design scales well when – (1) larger FPGAs are employed, or (2) greater I/O bandwidth to the system is available.

Fig. 3 shows the architecture. A linear array of $B$ PEs is used to perform FW on a $B$x$B$ matrix. Each PE has $l$ operators where each operator comprises a comparator and an adder. The $r^{th}$ PE stores the $r^{th}$ pre-computed pivot row and column, and the work it performs corresponds to the computation in the iteration $k = r$ of the outer loop of FW. The first and the last PEs read and write data respectively, from/to the I/O engine. The design is a streaming one, with read, compute and write pipelined. The latency to process a single tile is given by:

$$L \quad = \quad \left(\frac{3B^2}{l} + 3B - 1\right) \text{ cycles}$$

$l$ is the amount of *doAll* parallelism in each PE and is governed by I/O bandwidth. $B$ is the amount of pipelined parallelism and is constrained by the amount of FPGA resources. The product of $B$ and $l$ is the total parallelism available in our design, and this was maximized under resources constraints using a model.

The FW kernel described above can handle only matrices of size $B$x$B$. For the general case of $N$x$N$ matrices, we extend the kernel for the blocked algorithm proposed by Venkataraman et al. [13]. We provide a brief description of the same here.

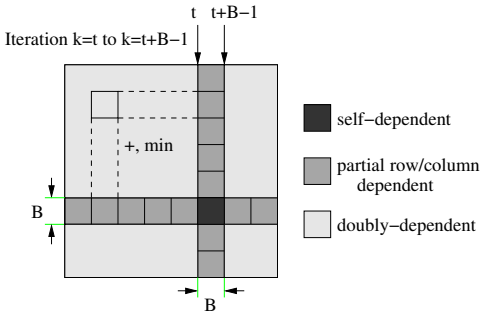


**Figure 4. Tiled FW algorithm**

Consider the sequential code in Fig. 2. At each iteration of the outermost loop, each element in the entire matrix is updated (if necessary) to the sum of its projections onto the pivot row and pivot column for that iteration. Now, consider the matrix to be composed of $B$x$B$ tiles as opposed to individual elements so that there are $(N/B)^2$ tiles, and consider a similar operation being performed on these 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 for a single round. 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). In each of the $N/B$ rounds, we process $(N/B)^2$ tiles using the FPGA design. Hence, a total of $(N/B)^3$ tiles are processed for an $N$x$N$ matrix. The design described in the previous section is used for processing self-dependent tiles; it was extended to process the other three kinds of tiles with minor changes.

**Table 1. FPGA-FW: Resource utilization and speedup on the Xilinx XC2VP50.**

| Tile size | FPGA-FW measured | CPU-FW | Measured speedup | Resource utilization |
|---|---|---|---|---|
| 8x8 | 0.42 $\mu$s | 1.60 $\mu$s | 3.8x | 34% |
| 16x16 | 1.29 $\mu$s | 14.1 $\mu$s | 11x | 52% |
| 32x32 | 4.84 $\mu$s | 106.5 $\mu$s | 22x | 90% |

The optimal values for $B$ and $l$ for the XD1 FPGA (Xilinx XC2VP50) were determined to be 32 and 4 respectively, for a precision of 16 bits. Table 1 shows the speedup of the FPGA kernel for processing a single tile.

## 4. Hardware/software Integration

In this section, we discuss some of the design and implementation issues in integrating the parallel FPGA kernel with software on the system.

### 4.1 Extending for any graph size

With the approach described in the previous section, we can handle a graph with a multiple of kernel tile size number of nodes. Any arbitrary size can be trivially solved by padding the distance matrix with $\infty$ (the largest value within the supported precision) to the next higher multiple of tile size. This is equivalent to adding additional disconnected nodes to the graph.

### 4.2 Data movement and communication

A specially allocated communication buffer that is pinned to the system's memory is used for transfer of data to/from the FPGA. The I/O engine (Fig. 3) on the FPGA transfers data of specified length to/from contiguous regions in the communication buffer. We split the buffer into two parts as shown in Fig. 5. The source and destination buffer addresses, amount of data to be transferred to/from these buffers, and the type of tile that is to be processed is communicated using a special set of registers. A write to the register meant for the destination buffer address triggers the computation on the FPGA. Completion is indicated by setting a register bit on the FPGA which the CPU polls. All of this overhead is incurred for each compute request made to the FPGA. Once the computation starts, all of the tiles placed in the source buffer are processed successively. As the computation for successive tiles is overlapped in the FPGA design (Sec. 3.3), the overhead associated with making the compute request is hidden when a large enough number of tiles are processed successively.

Allowing the FPGA to do the read and write from the communication buffer on the system's memory frees the CPU for other tasks. In particular, it provides us the opportunity to use the CPU to copy tiles between the matrix and the communication buffer while the FPGA is computing. We discuss this in Sec. 5.2.

The parallel FPGA kernel described in Sec. 3.3 was used iteratively taking care of dependences. All tiles of a particular kind (self-dependent, partial row/column

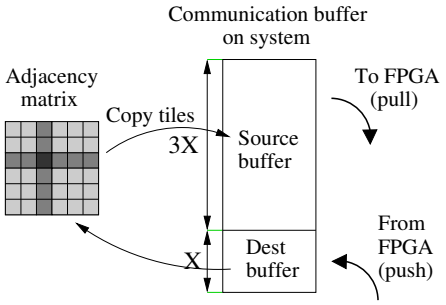


**Figure 5. Communication buffer model**

dependent and doubly-dependent) are processed successively. Hence, in each of the $N/B$ rounds of the blocked algorithm, the self-dependent tile is processed first followed by the sets of partially row and column-dependent tiles. This is followed by the processing of doubly-dependent tiles. In order to process a tile, the tile along with the pivot row and column elements is copied to the source buffer on the system interleaved in the fashion required by the kernel (Sec. 3.3). The compute request is then made to the FPGA. Hence, for a set of $k$ $B$x$B$ tiles that need to be processed successively, $3B^2k$ matrix elements are copied to the source buffer by the CPU. The FPGA reads from the source buffer and writes back the result tiles comprising $B^2k$ elements to the destination buffer, and sets the completion flag. The result tiles are then copied back to the matrix by the CPU. The number of copy operations for an $N$x$N$ matrix are therefore $4N^3/B$, where each operation involves a distance matrix element. It is the time consumed in these copies between the matrix and the source/destination buffers that we try to minimize.

## 5. Optimizing data movement

In this section, we discuss optimizations on the system-side to extract maximum performance from the FPGA kernel. The optimizations are partly model-driven and partly from analysis of measured performance of the unoptimized version which we refer to as FPGA-FW. In FPGA-FW, the three phases of copy, compute and write-back are done sequentially with the distance matrix in the original row-major form.

The entire blocked FW for an $N$x$N$ matrix comprises $(N/B)^3$ tiled computations. Each of the $B$x$B$ tiles in the matrix gets processed $N/B$ times – once in each round. Hence, temporal locality can be exploited for copying across rounds.

The copy time is proportional to the square of the tile size ($B$), while the number of compute operations is $\Theta(B^3)$. However, due to the FPGA kernel performing

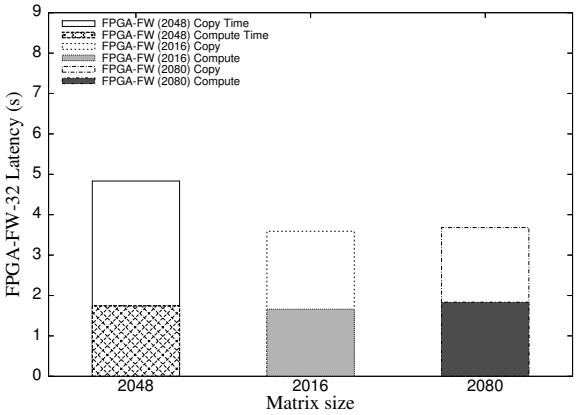


**Figure 6. Conflict misses in FPGA-FW-32**

these operations in parallel, the compute time may be reduced to a level where it is comparable to the copy time. This makes several approaches that would reduce or hide the data movement costs beneficial to pursue.

### 5.1 Layout transformation

In FPGA-FW, significant conflict misses occur as successive rows of a tile may get mapped to the same cache line due to size of the matrices being large and a power of two. Fig. 6 confirms this. This leads to loss of temporal locality across multiple rounds of the blocked algorithm. For column-wise copying, apart from temporal locality, spatial locality may be lost too. In addition, if the size of the cache line is greater than the size of a single row of the tile as is the case for 8x8 and 16x16 kernels, memory bandwidth is not effectively utilized. We therefore transform the layout of the input matrix so that $B$x$B$ tiles of the matrix are contiguous in memory in row-major order (Fig. 7).

After the layout transformation, the number of cache misses per $B$x$B$ tile for row-wise as well as column-wise copying is exactly $B^2/L$, where $L$ is the cache line size. This is true for large matrices that are at least twice as

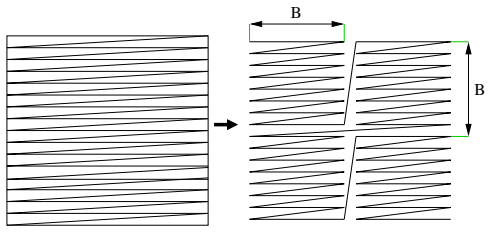


**Figure 7. Transforming the layout of the distance matrix**

large as the cache size. The cost of transforming the layout of the matrix is a small fraction of the FW latency and does not affect performance. Padding the distance matrix is another alternative, but is not as effective as layout transformation (Sec. 6).

### 5.2 Compute/Copy overlap

Even after performing the layout transformation, the compute time may be comparable to the copy time for matrices that do not completely fit in the cache. As we make the FPGA responsible for transferring data between itself and the communication buffer, the host is free while the FPGA is computing. We overlap the FPGA computation for a set of $k$ tiles with the write-back time and the copy time for the previous and next sets respectively. Two buffers are used in an alternating fashion (each for a maximum of $k$ tiles). We perform the compute/copy overlap only for doubly-dependent tiles as the processing of these tiles dominates the compute latency for the 32x32 FPGA kernel (Fig. 4).

**Optimal overlap chunk:** Choosing a small chunk size ($k$) for compute/copy overlap would not hide overhead that is involved in requesting the FPGA to process a set of tiles. Using a large chunk size would increase the trailing non-overlapping copy and write-back time (the first copy and the last write-back cannot be hidden). The optimal value for the compute/copy overlap chunk, $k$, is higher for larger matrices. We determine this in the next section.

## 6. Measurements and analysis

The measurements for the general-purpose processor case were taken on a 2.2 GHz 64-bit AMD Opteron (as found on the XD1) with a 64 KB L1 data cache and a 1 MB L2 cache with a cache-block size of 64 bytes. GCC 3.3 with "-O3" turned on was used for compilation. The FPGA on the XD1 is a Xilinx Virtex-II Pro XC2VP50. The FPGA design was clocked at 170 MHz. The version of Cray User FPGA API used was 1.3. The API provides functions to program the FPGA, write values and addresses to registers on the FPGA, taking care of virtual to physical address translation in the latter case. All measurements are for 16-bit edge weights. We use the following abbreviations to identify CPU and FPGA-based FW implementations with different optimizations throughout this section.

**CPU-FW**: Simple Floyd-Warshall (FW) implementation on the Opteron (three-way nested loop shown in Fig. 2).

**CPU-FW-OPT**: Optimized blocked implementation of FW on the Opteron (block size empirically optimized for best performance) [13]. We copy tiles to a contiguous buffer to eliminate conflict misses.

**FPGA-FW**: FPGA-based implementation for FW for an $N$x$N$ matrix on the XD1 FPGA without any optimizations.

**FPGA-FW-LT**: FPGA-FW with layout transformation as explained in Sec. 5.1.

**FPGA-FW-LTOV**: FPGA-FW-LT with compute and copy overlapped as explained in Sec. 5.2.

A suffix of 8, 16, or 32 is used to distinguish implementations using kernels that process tiles of that size. In all figures and tables, copy time refers to the sum total of both, the copy time and the write-back time. All speedups mentioned in this section are over the optimized CPU implementation (CPU-FW-OPT).

For graphs with up to 256 nodes, the distance matrix and the communication buffer completely fit in the L2 cache ($256^2$x2x5 bytes $<$ 1 MB). Hence, FPGA-FW, FPGA-FW-LT and FPGA-FW-LTOV perform equally well as seen in Table 2. However, there is a sudden increase in copy time from 256 nodes to 512 nodes, and performance drops from there on for FPGA-FW and FPGA-FW-LT (Fig. 10).

### 6.1 Effect of Layout Transformation

By operating on the bricked layout of the distance matrix, we find that the copy time for large graphs is cut down by more than two times. As shown in Table 2, the copy time for FPGA-FW-LT increases consistently by eight times as the size of the problem doubles, as opposed to the way it does for FPGA-FW. This is along expected lines (Sec. 4.2).

### 6.2 Effect of compute/copy overlap

Even after the layout transformation, for graphs with 512 nodes or more, the copy time is comparable to the compute time. The compute/copy overlap thus leads to a speedup by a factor of two hiding the copy time completely (Fig. 8). For the size of the overlap chunk (discussed in Sec. 5.2), empirically we find that a value of 32 works well for most problem sizes (Fig. 9). Thus, the total communication buffer requirements (including the alternating buffer) are: $2*4*k*B^2*2$ bytes $= 512$ KB.

For CPU-FW-OPT, the copy time is a very small fraction of the total latency (about 1%) for all problem sizes (Table 3). The compute and copy times for CPU-FW-OPT increase along expected lines (proportional to

**Table 2. Latency breakdown for 32x32 FPGA kernel with various optimizations**

| Size | FPGA-FW-32 | | | FPGA-FW-LT-32 | | | FPGA-FW-LTOV-32 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Compute | Copy | Total | Compute | Copy | Total | Compute | Copy | Total |
| 256 | 3.61ms | 1.41ms | 5.02ms | 3.66ms | 1.40ms | 5.07ms | 3.66ms | 0.43ms | 4.60ms |
| 512 | 27.86ms | 23.48ms | 53.36ms | 27.87ms | 22.58ms | 50.46ms | 28.68ms | 1.29ms | 31.05ms |
| 1024 | 0.22s | 0.21s | 0.43s | 0.22s | 0.19s | 0.41s | 0.23s | 0.01s | 0.24s |
| 2048 | 1.74s | 3.01s | 4.75s | 1.75s | 1.80s | 3.56s | 1.78s | 0.03s | 1.82s |
| 4096 | 14.02s | 28.75s | 42.77s | 14.02s | 14.40s | 28.43s | 14.39s | 0.14s | 14.55s |
| 8192 | 112.06s | 282.20s | 394.27s | 112.09s | 115.80s | 227.89s | 115.25s | 0.68s | 115.98s |

**Table 3. Measured performance: comparison with CPU-FW-OPT**

| Size | CPU-FW-OPT | | | FPGA-FW-LTOV-32 | | | Speedup |
|---|---|---|---|---|---|---|---|
| | Compute | Copy | Total | Compute | Copy | Total | |
| 256 | 69.6ms | 0.6ms | 70.2ms | 3.7ms | 0.4ms | 4.6ms | 15.2x |
| 512 | 480.5ms | 5.5ms | 485.9ms | 28.7ms | 1.3ms | 31.0ms | 15.7x |
| 1024 | 3.63s | 0.05s | 3.67s | 0.23s | 0.01s | 0.24s | 15.7x |
| 2048 | 28.04s | 0.33s | 28.37s | 1.78s | 0.03s | 1.82s | 15.6x |
| 4096 | 220.02s | 2.86s | 222.89s | 14.39s | 0.14s | 14.55s | 15.3x |
| 8192 | 1739.71s | 21.73s | 1761.44s | 115.25s | 0.68s | 115.98s | 15.2x |
| 16384 | 13810s | 171s | 3 hrs 53 min | 915s | 2.94s | 15 min | 15.2x |

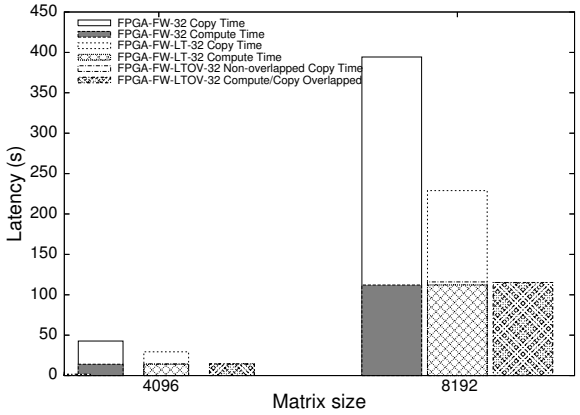


**Figure 8. Copy/compute time breakdown with different optimizations**

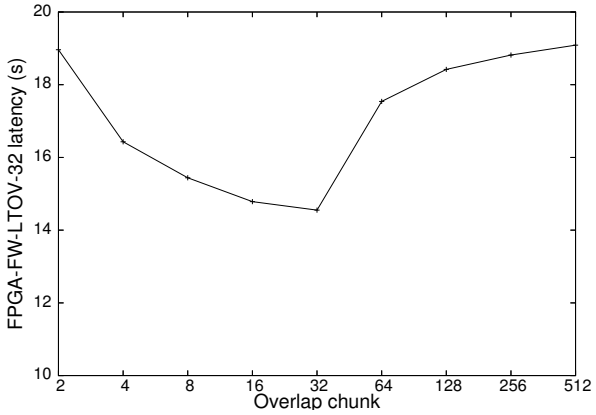


**Figure 9. Optimal compute/copy overlap chunk (for 2048 nodes)**

the cube of the problem size). As shown in Fig. 10 and Table 3, after all optimizations, a speedup of 15 is obtained for graphs with 256 nodes or more. For graphs with less than 256 nodes, the speedup is on the lower side due to the fact that there are not large enough number of tiles to be successively processed in each round to cover the overhead of making a compute request to the FPGA.

Fig. 11 shows the final speedup for all problem sizes obtained by employing the optimized 8x8, 16x16, and 32x32 FPGA kernels. The measured speedup doubles when the tile size that is processed in parallel doubles, as we have double the number of parallel operators then.

Fig. 12 shows the read bandwidth that the FPGA-based design obtains over the interconnect. For every $B$x$B$ tile that is processed, $3B^2$ 16-bit matrix elements are streamed to the FPGA, and $B^2$ matrix elements are written back. Read, compute, and write are fully

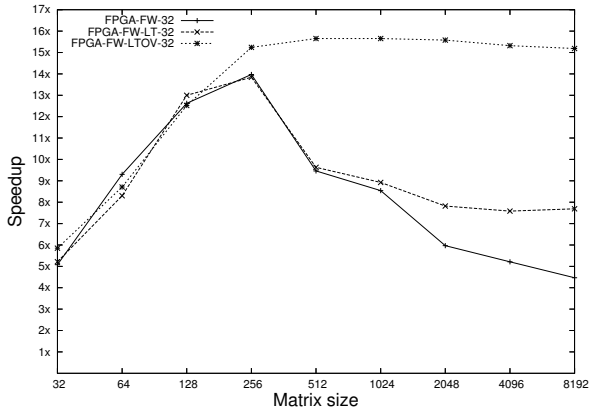


**Figure 10. Measured speedup over CPU-FW-OPT**

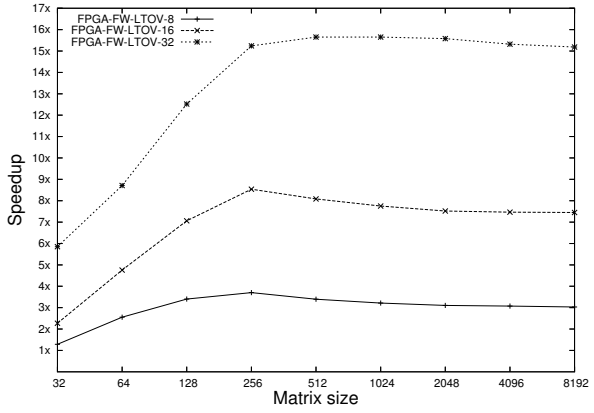


**Figure 11. Performance comparison: 8x8, 16x16, and 32x32 kernels**

pipelined. Hence, the read bandwidth that the FPGA kernel obtains is calculated as:

$$\text{Read b/w} = \frac{6N^3}{B * (\text{Compute time for } N \text{ nodes})} \text{B/s} \quad (1)$$

It is important to note that overlapping copying of tiles with the FPGA computation does not lead to a significant drop in memory bandwidth available to the host. As the FPGA computation contends with the copy operations for the memory bus (Fig. 1), the compute time for FPGA-FW-LTOV-32 is marginally higher than that of FPGA-FW-LT (Table 2).

### 6.3 Performance model

We build a model to project performance for larger FPGAs, and when higher bandwidth between the system and the FPGA

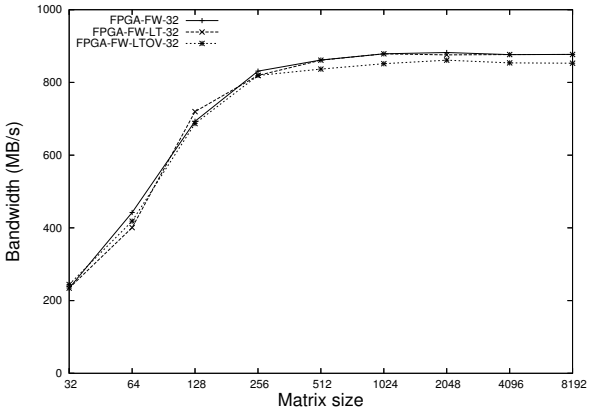


**Figure 12. Measured read bandwidth from the XD1 to the FPGA kernel**

is available. Let us define the following:

$t_{oh}$ : overhead for processing a set of tiles successively
$t_c$ : compute time for a $B$x$B$ tile
$t_{cp}$ : copy time (to source buffer) for a $B$x$B$ tile
$t_{wb}$ : write-back time for a $B$x$B$ result tile
$k$ : overlap chunk size (number of tiles)

As the overhead for making a compute request to the FPGA is incurred four times in each of the $N/B$ rounds (once for each type of tile), the latency for processing an $N$x$N$ matrix without overlap is given by:

$$L_{\text{no-ov}} = \left(\frac{4N}{B}\right) t_{oh} + \left(\frac{N}{B}\right)^3 (t_{cp} + t_c + t_{wb}) \quad (2)$$

We determine: $t_c$ = 6.68 $\mu$s, $t_{cp} + t_{wb} = 7.1$ $\mu$s, and $t_{oh} = 9.9$ $\mu$s, for $B = 32$. With compute and copy overlapped for doubly-dependent tiles, the latency is given by:

$$L_{\text{ov}} = \left(\frac{3N}{B} + \left(\frac{N}{B} - 1\right)^2 \frac{N}{B * k}\right) t_{oh} + k\,(t_{cp} + t_{wb}) + \left(\frac{N}{B}\right)^3 * \max\,(t_c, t_{wb} + t_{cp}) \quad (3)$$

Note that $t_c \propto (B^2/l)$ (Eqn. 1), and $t_{cp}$, $t_{wb} \propto B^2$. We compare the latencies obtained from the above equation with the measured ones for different problem and kernel sizes in Fig. 13 and Fig. 14 respectively. Note that $l$ is 4 for all the three kernels. Performance of the FPGA-based implementation is very accurately captured by the model.

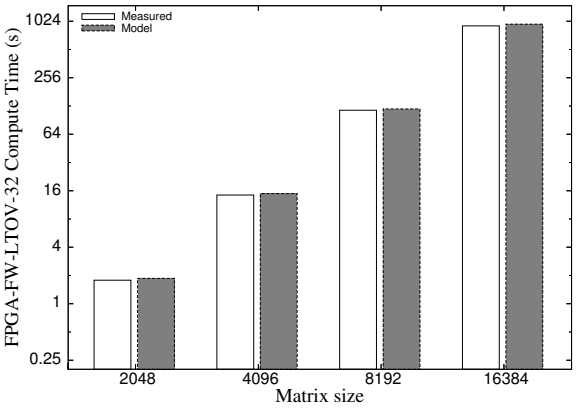


**Figure 13. Measured vs. Modeled latency for FPGA-FW-LTOV-32 (Eqn.3)**

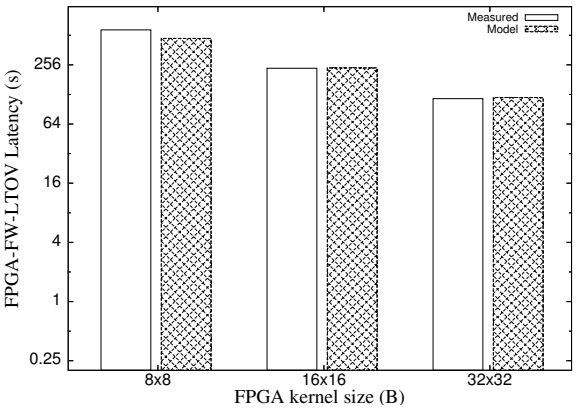


**Figure 14. FPGA-FW-LTOV latency for 8192 nodes with various kernels: Measured vs. Modeled (Eqn. 3)**

## 6.4 Impact of I/O bandwidth and FPGA resources

The number of element copies during FPGA-based FW on an $N$x$N$ matrix is $4N^3/B$. A larger tile size would process the matrix in terms of larger blocks, and so fewer copies. An increase in FPGA area would increase the tile size that would be processed in parallel ($B$). Thus, a higher value of $B$ would reduce both, the compute time (due to pipelined parallelism) and copy time. An increase in I/O bandwidth improves only the compute time as a result of higher *doAll* parallelism in each PE. Therefore, an increase in I/O bandwidth from the system to the FPGA would not help our application beyond a certain point where the compute time reaches the level of data movement costs on the system. In fact,

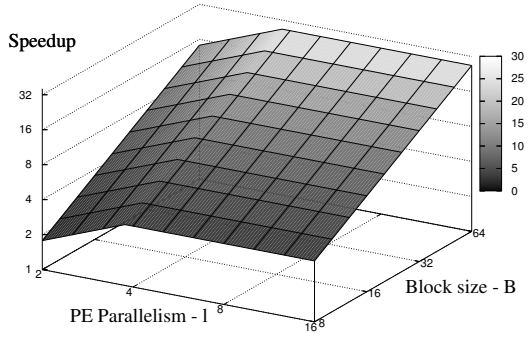


**Figure 15. Impact of FPGA-System Bandwidth and FPGA resources on FPGA-FW-LTOV for 8192 nodes: Eqn. 3**

with our design on the XC2VP50, and with the bandwidth we obtain, we reach this limit. We use the model built in the previous section to illustrate this effect in Fig. 15: greater I/O bandwidth beyond a certain point would be desirable only if more FPGA resources are available.

## 6.5 Performance improvement on a real dataset

Dynamic Transitive Closure Analysis (DTCA) is a recent algorithmic development for analysis of interaction and similarity networks of biological systems [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 [4, 7]. Reading in microarray expression data for several genes and drugs, the program utilizes 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 com-

puted co-correlation value.

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. We particularly consider a large instance of this problem – a graph with 22,740 nodes with FW called 100 times. The running time for this particular instance is 21 days on the Cray X1 (using eight vector pipes on an MSP), and 43 days on the Opteron. With FPGA-FW-LTOV-32, the running time is reduced to $2\frac{3}{4}$ days.

## 7. Conclusions

In this paper, we developed optimizations that enable high application-level speedup for blocked algorithms employing parallel FPGA kernels. Using a parallel FPGA-based Floyd-Warshall design developed in [1], the proposed optimizations minimize system-size data movement costs to effectively solve large instances of the all-pairs shortest-paths problem on a reconfigurable supercomputer. On the Cray XD1, these optimizations improve the application-level speedup of the FPGA-based implementation from 4 to 15. A model was developed to accurately characterize the latency of the optimized FPGA-based implementation and provide an insight into the impact of memory bandwidth and FPGA resources on the achieved speedup. The techniques developed apply to other blocked FPGA-accelerated routines for which the speedup with the FPGA kernel is high enough to make orchestrating movement of data on the system crucial. With the current trends in FPGA and CPU performance, these approaches would become increasingly important for most FPGA-based routines on reconfigurable supercomputers.

## References


[1] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel FPGA-based All-Pairs Shortest-Paths in a Directed Graph. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, Apr. 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. http://www.cray.com/products/xd1/.

[4] 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.

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

[6] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. In *IEEE Symposium on Foundations of Computer Science*, pages 560–568, 1991.

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

[8] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *IEEE Symposium on Foundations of Computer Science*, pages 605–615, 1999.

[9] SRC Computers Inc. SRC MAPstation. http://www.srccomp.com/MAPstations.htm.

[10] J. L. Tripp, A. A. Hansson, M. Gokhale, and H. S. Morveit. Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study. In *Proceedings of ACM/IEEE Supercomputing*, Nov. 2005.

[11] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field-programmable Gate Arrays*, pages 171–180, 2004.

[12] 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*, pages 219–228, Apr. 2004.

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

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

[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.

[17] L. Zhuo and V. K. Prasanna. Sparse Matrix-Vector multiplication on FPGAs. In *FPGA '05: Proceedings of the ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, pages 63–74, 2005.