# Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications

Keith D. Underwood* K. Scott Hemmert†
Craig Ulmer‡
Sandia National Laboratories§
P.O. Box 5800, MS-1110, Albuquerque, NM 87185-1110

## Abstract

Reconfigurable computing leveraging field programmable gate arrays (FPGAs) is one of many accelerator technologies that are being investigated for application to high performance computing (HPC). Like most accelerators, FPGAs are very efficient at both dense matrix multiplication and FFT computations, but two important aspects of how to deliver that performance to applications have received too little attention. First, the standard API for important compute kernels hides parallelism from the system. Second, the issue of system architecture is virtually never addressed. This paper explores both issues and their implications for applications. We find that high bandwidth, low latency connectivity can be important, but the right API can be even more important.

**Keywords:** IEEE floating point, FFT, Matrix Multiply, FPGA, reconfigurable computing

## 1 Introduction

Despite renewed interest in accelerator technologies for use in high performance computing (HPC), there has been relatively little research on the broad system implications of this technology. FPGAs are an excellent example. FPGAs are known to greatly accelerate a large class of applications; however, the classic usage model for FPGAs is to select a portion of the application that consumes a large percentage of the overall execution time and minimizes the necessary I/O. This has been driven by the traditionally loose coupling between host processors and accelerator boards. It does not make sense to tightly couple resources across a PCI bus. Unfortunately, traditional HPC applications are not amenable to this model.

Many traditional HPC applications fall in the domain of scientific computing. While there are exceptions (notably molecular dynamics (MD)[Kindratenko and Pointer 2006; Scrofano et al. 2006]), these applications tend to be hundreds of thousands of lines of code without a small section that dominates execution time. This led researchers to begin by focusing on kernel operations that are used in HPC and can be provided through a standard library interface. Operations from BLAS [Underwood and Hemmert 2004; Zhuo and Prasanna 2004; Dou et al. 2005; Zhuo and Prasanna 2005a; Zhuo and Prasanna 2005b] to FFTs[Hemmert and Underwood 2005] to the sparse matrix operations at the core of an iterative solver[deLorimier and DeHon 2005; Zhuo and Prasanna 2005c] and even a full CG solver[Morris et al. 2006] have been studied. The fundamental challenge for each of these efforts is the communications with the host. Accelerators cannot receive data faster than the host processor's memory bandwidth, but they are often further constrained by the I/O bus. Overcoming the I/O limit typically requires large operations, but large, dense operations are seldom used in scientific applications. More common are large numbers of small, dense operations. For example, Climate and MD codes perform a large number of small 1D-FFTs and some quantum chemistry codes perform a large number of small matrix multiplies. Even some solvers have been adapted to use many calls to small, dense matrix multiplies. This has lead many to question whether an accelerator can accelerate scientific computations.

Given the growth in vendors offering accelerators from Clearspeed parts to FPGAs, there are major questions that need to be addressed regarding how applications can use these accelerators. For example, how much bandwidth is needed by the accelerator? How much does latency matter? Will our current APIs work? Even the best technologies can fail if these issues are incorrectly addressed.

This paper examines the complex double precision FFT and double precision, dense matrix multiplies (DGEMM) oper-

*e-mail:kdunder@sandia.gov
†e-mail:kshemme@sandia.gov
‡e-mail:cdulmer@sandia.gov
§Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.


SC2006 November 2006, Tampa, Florida, USA

Table 1: Typical system parameters

| System Type | Latency | Bandwidth |
|---|---|---|
| PCI-X | 1000 ns | 1000 MB/s |
| PCI-Express 8X | 800 ns | 3.6 GB/s |
| HyperTransport (HT) | 250 ns | 3.2 GB/s |
| Memory | 250 ns | 3.6 GB/s |

ations in the context of the way they are often used: large numbers of small operations. It begins by analyzing the implications of the traditional blocking function call found in traditional libraries. Then, experimental results from the Cray XD1 platform demonstrate that even FPGAs with modest interconnections to the host can provide measurable wins over a microprocessor when using a better API. System level simulations extend this work to explore the requirements for next generation FPGAs. The findings indicate that both the architecture and the API are critically important. High latency, low bandwidth interfaces can clearly be limiting factors, but no realistic system is going to provide a low enough latency, high enough bandwidth interface to cope with the blocking procedure calls that are typically used to interface to today's libraries.

The next section discusses the motivating applications and architectures behind this work, along with the methodology used. In Section 3, the nonblocking approach is presented and contrasted with the traditional approach. Following that, Section 4 presents an analysis of the impacts of the latency and bandwidth of the connection between the FPGA and the host processor. Sections 5 and 6 present measured results from the Cray XD1 and a system simulation environment, respectively. The paper concludes with related work (Section 7) and conclusions (Section 8).

# 2 Motivation and Methodology

There has been a recent surge in work on new system architectures supporting FPGAs (and accelerators in general). Each system has different implications for the aggregate bandwidth and latency of the connection to the processor (Table 1[1]). While a large body of work has explored the *potential* of these systems for scientific computing, little work has discussed the context of real applications. Research tends to explore the power of FPGAs to address large dense matrix algorithms (DGEMMs) and large FFTs, but such work seldom comments on any applications that actually use such operations (mostly because such applications are rare). More importantly, few analyses have looked at the salient properties of the architectures to determine their relative importance.

[1]HyperTransport (HT) can achieve over twice the listed rate, but it is constrained by FPGA I/O capabilities.

## 2.1 Motivating Applications

Generally speaking, scientific applications do not leverage large dense matrix operations. Indeed, we know of no supercomputer applications at Sandia that leverage them. Small dense matrix operations are a different story. For example, a large number of small, double precision, complex 1D Fast Fourier Transforms (FFTs) is used in parallel molecular dynamics codes (e.g. LAMMPS[Plimpton 1995; Plimpton et al. 1997]) as part of a distributed 3D FFT. The typical approach to a distributed 3D FFT is to perform three iterations of $N^2$ FFTs of size $N$ with data transpositions between the iterations. The typical size of a call is on the order of 128 to 256 and $N^2$ consecutive, independent calls to the function are made. Similarly, forecasting codes can use thousands of small (size 256 to 1500), consecutive, independent 1D FFT calls[Williamson et al. 1992]. FFTs are a significant, although not dominant, contributor to overall run-time for these applications.

Another example is the dense matrix multiply operation (DGEMM). While none of Sandia's applications use DGEMM operations for large operations (dimensions of hundreds or more), many of them do small, dense, double precisions matrix multiplies. For example, MPQC[Janssen ] is a quantum chemistry code that uses thousands of small, independent matrix multiplies ranging from $15 \times 15$ to $50 \times 50$. Similar patterns occur in sparse matrix multiplies that are used in Mondo SCF (another quantum chemistry code). Finally, modern solver techniques are moving to locally dense, globally sparse matrices with multiple right-hand sides. This also leads to a large number of independent DGEMM operations that tend to be approximately $16 \times 16$. It is anticipated that the DGEMM calls will become as much as 90% of the execution time as other aspects of the codes are tuned.

## 2.2 Methodology

This effort used two hardware platforms and a system level simulator to run the same benchmarks. The first hardware platform was the Cray XD1, which was used as both the FPGA platform and as a processor platform to measure the performance of the Opteron. The second hardware platform was a Pentium-4 Xeon workstation that was used as an example of the *best available* processor performance, since the Pentium-4 is known to have a higher peak floating-point rate on dense matrix operations than the Opteron.

### 2.2.1 Cray XD1

Figure 1 depicts an XD1 compute blade containing two AMD Opteron processors. One CPU connects directly to a network interface (NI) chip using a HyperTransport (HT) link. This NI uses a Xilinx Virtex2Pro FPGA that limits the

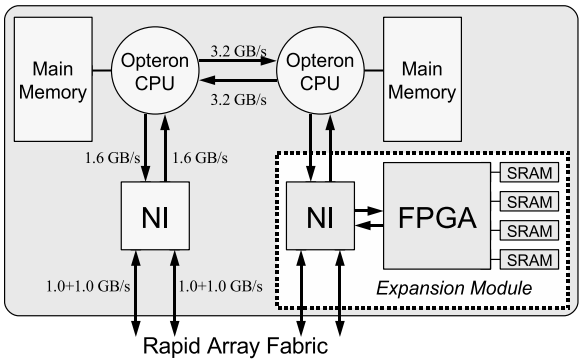


Figure 1: The XD1 architecture

CPU to network interface link to 1.6 GB/s per direction (1.4 GB/s after overhead). The user-programmable FPGA is on an expansion board connected to an HT interface on the second processor. As Figure 1 illustrates, the user FPGA is connected to a second NI through a simplified version of HT. This interface enables the FPGA to read and write the hosts memory, as well as respond to memory requests issued by the host processor. The XD1 utilized for this work is populated with V2P50-7 expansion boards and Opteron 248 processors, which are clocked at 2.2GHz and have a peak of 4.4 GFLOPs.

The FPGA design flow used Xilinx's ISE 6.3.03 tool chain including the Xilinx Synthesis Tool (XST) for VHDL synthesis. The XD1 system runs the 1.1 release of Cray's system software with a modified Linux 2.4.21 kernel. The 1.1 release of the XD1 system software can only pin one 2MB block of contiguous host memory for sharing data with the FPGA. This typically requires a data copy, but the more recent 1.3 release of XD1 system software uses the Graphics Address Resolution Table (GART) to address up to 1 GB of host memory from the FPGA and eliminates the need for a copy. Our tests emulate this behavior by not performing the final copy in the 1.1 release.

#### 2.2.2 Pentium-4 Xeon Workstation

A Pentium-4 Xeon workstation was used as one of the processor baselines for comparison. The workstation has two 3.2 GHz Pentium-4 Xeon processors with EM64T technology. It is equipped with 8 GB of RAM and is running the RedHat Enterprise 4 WS Linux distribution. The compiler used was GCC 3.4.5, but the compiler only compiled the outer loop. Matrix operations were measured using FFTW version 3.1[Frigo and Johnson 1998] and the Intel Math Kernel Library (MKL) version 8.1. FFTW was faster for the FFT portion of the test, and so it was reported in the results section.

#### 2.2.3 Structural Simulation Toolkit

To explore configurations beyond the XD1, we used the Structural Simulation Toolkit (SST). The Structural Simulation Toolkit is built around Enkidu, a hybrid simulation framework that optimizes for the common case in architectural simulation by providing low-overhead synchronous time-stepping to handle most functionality. For less frequent communication between components, an asynchronous event mechanism is provided. SST integrates the SimpleScalar (v3.0) toolkit's sim-out-order processor model[Burger and Austin ] to model conventional processors. For these experiments, an execution-based front end supporting PowerPC Mach-O binaries was used.

### 2.3 Benchmarks

Two benchmarks were used to test these concepts and provide a comparison point against traditional microprocessors. These simple benchmarks were designed to capture the way applications work; thus, they differ slightly from a traditional benchmark. A comparison is shown in Figure 2 — the difference is subtle, but critical. Where a typical benchmark does a large number of iterations over a single buffer, real applications tend to do a large number of iterations over *different* buffers. Thus, the benchmarks reflect that.

## 3 Approach

Virtually every major HPC system shipped today requires standard libraries to be available for the Basic Linear Algebra Subroutines (BLAS) and FFT. Some vendors have proposed using hardware accelerators to intercept these calls and, thus, provide improved performance. The problem, however, is in the traditional semantics of a blocking subroutine call. Contrast the examples in Figure 3. The blocking calls are perfectly suitable for either execution on a host microprocessor or for performing large routines on a compute accelerator; however, when there is a large number of small operations to do, the nonblocking calls expose more parallelism to the system and enable the system to pipeline these operations. The net effects can be seen in the timeline in Figure 4, where the nonblocking calls can leverage the double buffering on the accelerator to overlap communication between the host processor and accelerator with computation that is occurring on the accelerator. While this is conceptually straightforward, this paper aims to quantify the impact in the context of a specific technology to motivate the development of appropriate APIs.

To target the way Sandia's applications use DGEMM and FFT calls, we focus on implementations that can accelerate large numbers of small calls. Thus, rather than one large unit

```
choose_operation_size();
buffer=allocate_buffer();
begin_timer();
for (i = 0; i < N; i++)
    do_op(buffer);
end_timer();
```

**(a)**

```
choose_operation_size();
for (i = 0; i < N; i++)
    buffer[i] = allocate_buffer();
begin_timer();
for (i = 0; i < N; i++)
    do_op(buffer[i]);
end_timer();
```

**(b)**

Figure 2: The traditional (a) and modified (b) benchmarks

```
choose_operation_size();
for (i = 0; i < N; i++)
    buffer[i] = allocate_buffer();
for (i = 0; i < N; i++)
    do_op(buffer[i]);
```

**(a)**

```
choose_operation_size();
for (i = 0; i < N; i++)
    buffer[i] = allocate_buffer();
for (i = 0; i < N; i++)
    start_op(buffer[i], request[i]);
wait_all(request, status);
```

**(b)**

Figure 3: **(a)**Blocking vs. **(b)**Non-blocking approaches

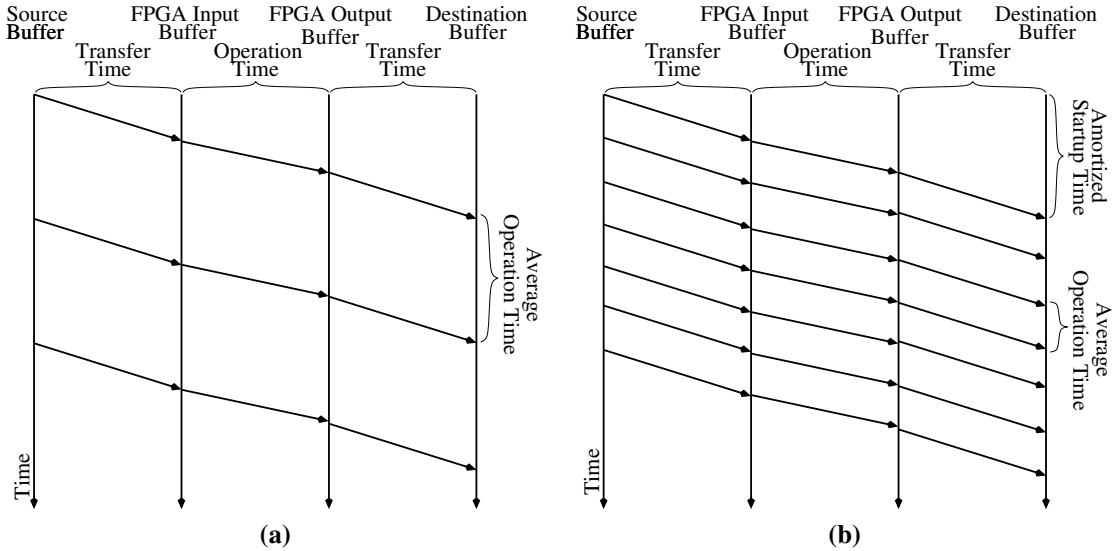


Figure 4: Timeline for **(a)** blocking and **(b)** non-blocking operations

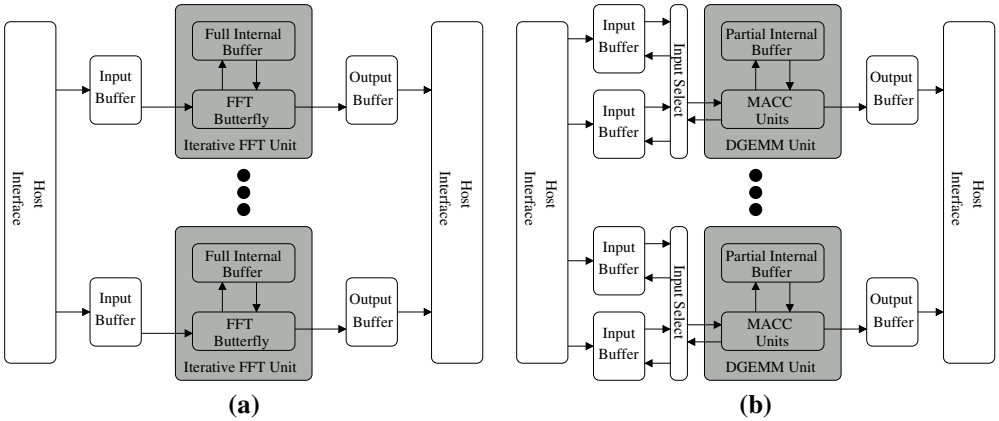


Figure 5: Buffer approaches for **(a)** FFT and **(b)** DGEMM.

in the FPGA, we have several smaller units that can each handle the processing for a call. These are integrated with the host using double buffering techniques so that data can be transferred between the host and FPGA while the computation is occurring. Figure 5 illustrates this concept for the FFT and DGEMM operations.

The FFT and DGEMM operations in Figure 5 take slightly different approaches. While the FFT needs *both* ports of a dual-ported RAM during the core operation, the DGEMM operation does not. Therfore, the FFT uses full sized input, output, and internal buffers. This is sufficient to provide double buffering for the FFT. The DGEMM operation, on the other hand, allows the system to manage the double buffering. An input selection mechanism makes this transparent to the unit. In practice, this is implemented by splitting the top and bottom of a single dual-ported RAM bank into independent buffers and selecting based on the high order address bit.

# 4 Analysis

Dense matrix operations are regular enough to lend themselves to direct analysis. Similarly, cycle accurate simulation of hardware level details makes it possible to reliably capture hardware overheads. Thus, it is possible to create analytical models that accurately reflect major portions of expected system performance.

## 4.1 Fast Fourier Transform

The FFT operation requires $5Nlog_2(N)$ operations for a complex FFT of size $N$, which requires $16N$ bytes of input and $16N$ bytes of output. Using a non-blocking implementation for a stream of independent FFTs, this can be implemented with several, independent iterative butterfly units (like the units used in [Hemmert and Underwood 2005]) that perform $5N$ computations in a single pass over the data requiring $N$ cycles. The unit must perform $log_2(N)$ passes and has an initialization latency of 51 cycles. Thus, from the time the data is delivered, the hardware needs:

$$T_F = Nlog_2(N) + 51 \tag{1}$$

cycles to compute one FFT, and has been verified through cycle accurate simulation at the HDL level. Since $P$ of these units run concurrently, the minimum average time per FFT is:

$$T_F = \frac{Nlog_2(N) + 51}{P} \tag{2}$$

A blocking implementation must leverage all of the FPGA for a single operation. The best design for the domain of interest is a fully parallel layout of $P$ butterfly units, as described in [Hemmert and Underwood 2005]. The time through this unit was derived to be:

$$T_F = \frac{2N}{BW} + BL + (\frac{N}{P} + BL)(log_2(N) - 2) \tag{3}$$

This assumes an FFT of $N$ elements and a local memory bandwidth, $BW$, that is given in "complex double precision floating-point items per cycle". $BL$ is the latency through a single butterfly unit (the initialization latency) and $P$ is the number of units used. Only $log_2(N) - 2$ iterations are required, because the first and last iteration happen during input and output. With sufficient FPGA resources, "lead-in" units (details available in [Hemmert and Underwood 2005]) can reduce the total time to:

$$\begin{aligned} T_F &= \frac{2N}{BW} + BL + Startup + \\ &\quad (\frac{N}{P} + BL)(log_2(N) - (2 + log_2(P))) \end{aligned} \tag{4}$$

where $Startup$ is the latency through those "lead-in" stages. An important caveat, however, is that the high latency of the floating-point units addes significant performance penalties for sizes under $P * 64$ elements. These penalties are accounted for in the simulations, but are not presented here.

Integrating this design with this host is more complex. The access patterns of the FFT computation are not compatible with typical bus interfaces. In these examples, the butterfly operation is reversed from some traditional approaches such that the first accesses are $N/2$ items apart. Thus, most systems will need to transfer the entire buffer from the host to the FPGA before the FFT computation can begin and will then need to build the entire buffer before it can be transferred to the host[2]. Thus, a blocking implementation adds the time to transfer the data to the time to do the operation:

$$T_{Tot} = \frac{2N}{HostBW} + T_F + 2 \times HostLat \tag{5}$$

Again, the bandwidth to the host, $HostBW$, is given in complex, double-precision items per second, and the latency to the host is $HostLat$.

In a non-blocking implementation, the data is double-buffered. This means that the operation can be overlapped with both the transfer of the result from the last operation to the host and the data for next operation from the host. The

[2]This is not strictly true. A processor connected by HT could probably transfer data in arbitrarily placed 64 byte chunks allowing *some* computation to progress while the data was being transferred. The impact is negligible for these examples.

three dependent stages become three independent stages so that the time per operation drops to the maximum of the time to transfer the data (once) and the time to perform the operation:

$$T_{Tot} = max(\frac{N}{HostBW}, T_F) \quad (6)$$

This is a major improvement and can be a factor of three at operation sizes of interest. The pipelining of the communications also amortizes the cost of the latency between the host and the FPGA across all of the FFT calls. When assuming hundreds of calls, it is eliminated.

## 4.2 Dense Matrix Multiply

The basic DGEMM[3] routine performs the operation $C = A \times B + C$. From the perspective of an accelerator, this requires $2N^3$ operations using 3 input matrices of $N^2$ words and one output matrix of $N^2$ words. The small matrix multiplies are implemented with an array of multiply-accumulates (MACCs), as described for large matrix multiplies in [Underwood and Hemmert 2004]. In principle, the DGEMM operation is compute bound, since it performs $2N^3$ operations over only $3N^2$ data, with $4N^2$ memory operations. Thus, the operation should require:

$$T_{MM} = \frac{N^3}{M} \quad (7)$$

cycles, where $M$ is the number of MACCs used; however, practical constraints lead to an initialization latency. Furthermore, to simplify the implementation, the final output was not overlapped with the next operation. Thus, the real time becomes:

$$T_{MM} = IL + \frac{N^3}{M} + OL \quad (8)$$

where the initialization latency, $IL$, is approximately 25 cycles in this implementation and includes time to setup parameters, retrieve the data from block RAM, propagate it to the right unit, and propagate through the multiply. The output latency, $OL$, is tied to the concurrency needed in the MACC unit[4] and the bandwidth of the memory associated with the output. In our implementation, that bandwidth is one double precision floating-point item per cycle leading to $OL = M * 16$, where 16 is the concurrency used for each MACC unit. Each of these numbers have been verified through cycle accurate VHDL simulation.

[3] We will ignore things like the application of a scalar and transposes of the matrices for now.

[4] The long adder latency requires that several multiply-accumulates run concurrently.

As with the FFT, a group of MACCs is part of a larger processing unit. When $P$ of these processing units are used, the minimum average time per DGEMM operation drops to:

$$T_{MM} = \frac{IL + \frac{N^3}{M} + OL}{P} \quad (9)$$

Integration with the host has very similar properties to the FFT. Due to the access patterns of the $B$ matrix, the data must be transfered down in its entirety before the computation can begin[5]. Thus, the time is:

$$T_{Tot} = \frac{4N^2}{HostBW} + T_{MM} + 2 \times HostLat \quad (10)$$

The $HostBW$ term in Equation 10 is given in double-precision floating-point items per cycle in this case. Double-buffering with a non-blocking implementation significantly reduces the time per operation:

$$T_{Tot} = max(3N^2 HostBW, T_{MM}) \quad (11)$$

Unfortunately, the bandwidth needs of the DGEMM operation are asymmetric as it reads three times as much data as it writes. As with the FFT, we have amortized away the latency cost.

## 4.3 Implications for Architecture

The analysis in this section does not account for potential sources of overhead, ranging from bus contention to the operating system to the software stack. Instead, they provide a *best-case* scenario for analyzing the important aspects of FPGA based system architecture. Figure 6 considers the impact of bandwidth and latency on small, blocking FFT and DGEMM operations. The figures present data for 8 GFLOPs and 6.4 GFLOPs peak designs, respectively, representing the best case for a Xilinx Virtex2Pro100-6. Each graph presents two input sizes. The larger size stretches the bounds of what might be useful for the applications discussed in Section 2 and for most scientific applications in use at Sandia National Labs. The smaller size is representative of what might be considered "typical" in Sandia's applications.

Even at small data sizes, Figure 6 makes it clear that the biggest issue for application usage in this domain is bandwidth to the host processor. Only the smallest operations with a high bandwidth connection show an appreciable difference when the latency of that connection is quadrupled from 250$ns$ to 1000$ns$.

In contrast, the nonblocking operations are completely insensitive to latency, because they assume that numerous independent operations will be done. The first observation

[5] Or, an expensive transpose of $B$ must be done on the host.

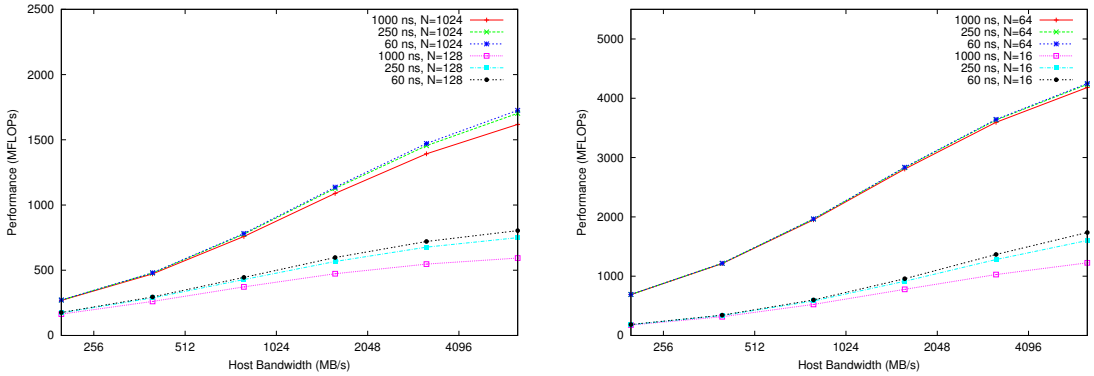


Figure 6: Analytical impacts of bandwidth and latency on small, blocking **(a)** FFT and **(b)** DGEMM

from comparing Figures 6 and 7 is the stark contrast between the achievable performance with blocking operations and nonblocking operations, but Figure 7 offers other interesting insights. Like many dense matrix operations, the FFT and DGEMM operations demonstrate a superlinear growth in work as the input size is increased. Despite this, the small operation sizes used by many scientific applications can easily leave an accelerator bandwidth bound. Indeed, it is not until 3.2 GB/s of bandwidth that an 8 GFLOPs accelerator can approach its peak performance for a small $Nlog(N)$ operation like the FFT. In contrast, the DGEMM operation is an $N^3$ operation. Though this implies *more* work per $N$, the data grows as $N^2$ and $N$ is relatively small. While an $N$ of 64 can achieve peak performance with only 1.6 GB/s, matrices with a dimension of 8 do not achieve the peak performance of a 6.4 GFLOPs accelerator with 6.4 GB/s of bandwidth to the host.

# 5 Measurements from the XD1

To validate our analysis and simulations and to provide a concrete comparison point between microprocessors and FPGAs, we took measurements from the Cray XD1 and from commodity microprocessors. Figure 8(a) compares the FFT performance of the FPGA on the Cray XD1 to the Opteron on the same platform and a recent Pentium-4 Xeon processor. The first observation about the graph is that it is very different from the "standard" benchmark graphs for the processors used, because our benchmarks performed 1000 operations over 1000 different buffers where traditional benchmarks perform 1000 operations over the same buffer.

The next important note is that the FPGA on the Cray XD1 can achieve a 60% to 125% performance improvement, depending on problem size, if a nonblocking API is used. Using a blocking API stands in stark contrast with 60% *lower* performance than the microprocessor. Results from the Pentium-4 Xeon indicate that this 2.5 year old FPGA that is only half as big as the largest FPGA from that era can outperform one of the fastest current single core processors — if, and only if, it uses a nonblocking interface.

Figure 8(b) presents results from DGEMM and contrasts with the FFT results. Modern microprocessors are highly optimized to perform operations like DGEMM; thus, the FPGA loses by as much as a factor of 2, even when it uses a nonblocking interface. Given that it is a relatively old, relatively small FPGA, this is not surprising.

# 6 Simulation Results

Like microprocessors, FPGAs have reaped many benefits from Moore's Law. In fact, recent FPGA performance gains have outstripped the performance gains of microprocessors[Underwood 2004]. To explore the near-term potential of FPGAs, we leveraged a hybrid discrete-event/cycle-driven simulator. The initial simulations were used to validate the simulator. Figure 9 indicates that the simulator captures most of the salient points of the system. For the FFT, the only point that is not in almost perfect agreement (within 1%) between the simulator and the XD1 implementation is the 128 point non-blocking FFT. The simulation is more representative of what can be achieved, but there is approximately 500 ns of overhead in the XD1 implementation that is associated with buffer management that we are trying to eliminate.

The DGEMM results are slightly less tightly correlated. For many cases, the simulation results are within 1% of the measured values. From an $N$ of 20 to 32, the results are within 5%. Measurements indicate that at least a 1% difference can be attributed to the inaccuracies in the actual clock rate on the XD1. Below an $N$ of 16, the operations use smaller transfers than the 128 point FFT; thus, the differences occur for the same reason — unexplained overhead in the interaction with the host that is being debugged. Blocking simulations were

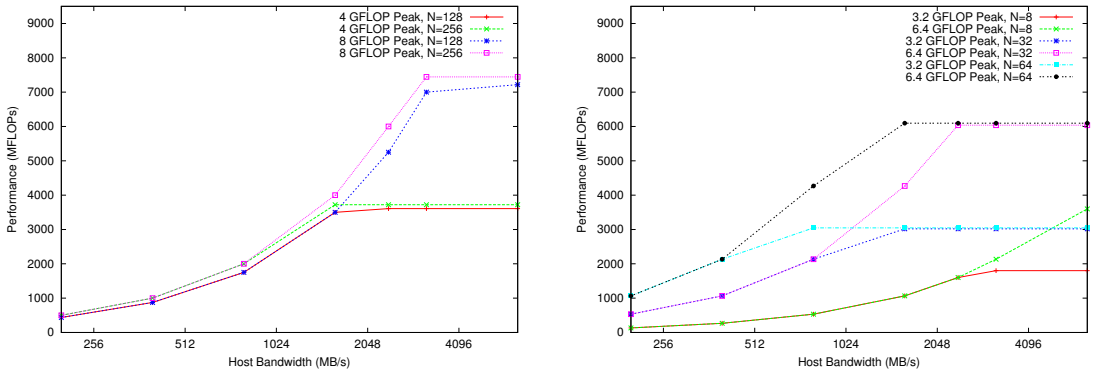


Figure 7: Analytical impacts of bandwidth on small, nonblocking **(a)** FFT and **(b)** DGEMM

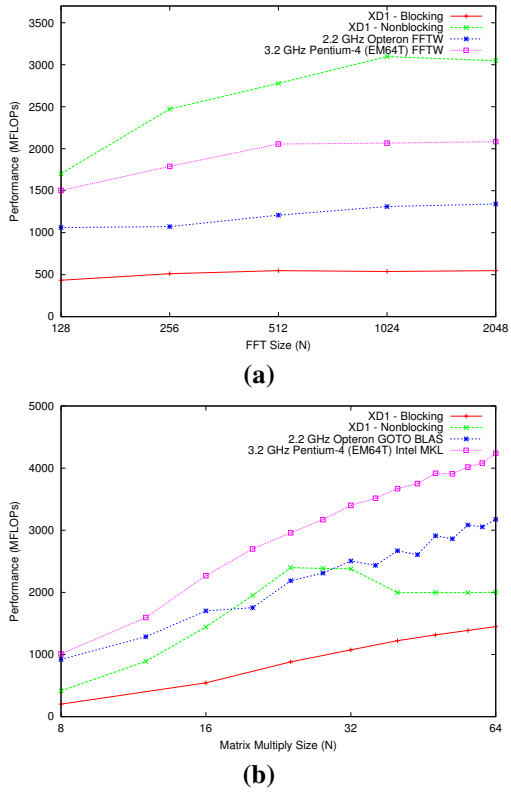


Figure 8: Comparison of FPGA performance to processor performance for **(a)** FFT and **(b)** DGEMM as measured on the XD1

only run over the range of $N$ from 32 to 64, since those are the only points where the blocking implementation fully utilizes the resources. For these runs, the simulations are within 3% of the measured values. Overall, we believe the simulations provide sufficient predictive capabilities when the predicted differences are large.

The second set of simulations focuesd on the bandwidth and latency requirements for FPGAs now and as FPGAs increase in performance. These simulations used nonblocking operations and swept latencies from a minimum of 50 ns to a maximum of 1000 ns. With nonblocking operations, latency had no impact on the performance and those results are omitted for brevity. FPGA sizes are modeled based on the Virtex2Pro50-7 on the Cray XD1, the Virtex2Pro100-6 available on SRC systems, one Moore's Law doubling (40 MACC units, 250 MHz, roughly a Virtex4-FX140) and two Moore's Law doublings (80 MACC units, 380 MHz)[6]. Results presented in Figure 10 assume a latency between the processor and FPGA of 250 ns.

For currently available systems (Figures 10 (a) and (c)), the choice of device matters. Larger devices deliver more performance with bandwidths that are readily achievable. It is also clear that PCI-X levels of bandwidth (500 MB/s/direction or 1000 MB/s total) are not sufficient for the domains of interest discussed in Section 2, and, generally speaking, modern FPGA systems need more bandwith. Even the bandwidth of HT (1400 MB/s/direction sustained) is not quite sufficient.

A more interesting story arises from Figures 10 (b) and (d). All but one pair of lines completely overlaps with the generation after it all the way to 5 GB/s/direction. Two parts with a 3× difference in performance are both bandwidth constrained to the same performance in the domain of interest. A more aggressive design point could take the bandwidth to the FPGA up to 10 GB/s/direction or even 20 GB/s/direction as

[6]These numbers assume that a generation gives 2× the transistor count, but somewhat under 2× the clock rate. Clock rate growth is difficult to forecast for floating-point on FPGAs as it is virtually independent of technology in the near term and depends on FPGA features introduced.

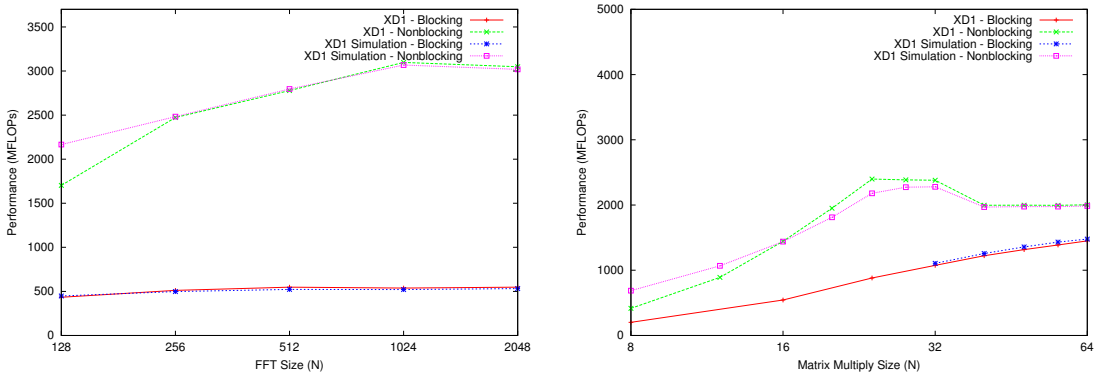


Figure 9: Validation of the simulation models for **(a)** FFT and **(b)** DGEMM

technologies like HT-3 come online. While the graphs show the potential for strikingly high levels of performance, the I/O connections to FPGAs will have to improve dramatically to leverage any of those potential gains.

While the bandwidth requirements seem extreme, there are bright points. Foremost, it is unlikely that the largest FPGAs (represented by the "two doublings" line) will ever be cost-effective for HPC applications. With the performance of FPGAs growing faster than that of microprocessors[Underwood 2004], this would indicate that a smaller, cheaper FPGA could deliver all of the performance that the bandwidth will support. Also, with growth in computing power comes growth in the amount and types of science that can be done. This tends to increase the size of operations such that $16 \times 16$ DGEMMs and 128 point FFTs are likely to become $32 \times 32$ DGEMMs and 512 point FFTs in another generation. These trends will somewhat reduce the bandwidth needed to achieve higher levels of FPGA performance.

The final set of simulations explored the impact of a traditional blocking API on FPGA performance at two generations beyond the Virtex2Pro100. The latency between the processor and the FPGA was set to an optimistic 50 ns (to offer every advantage to the blocking approach) and bandwidth was set to 5 GB/s/direction to match what should be achievable soon with HT or PCI-Express. Assuming that at least 1000 independent operations are required yields the graphs in Figure 11. The blocking calls are $2\times$ to $3\times$ slower than their nonblocking counterparts. More importantly, the blocking graph for the matrix multiply *starts* at $32 \times 32$ because this is the first point at which a matrix multiply could fully utilize the device.

# 7 Related Work

The intersection of traditional high performance, scientific computing and FPGA based reconfigurable computing only recently began to mature. Studies have indicated that FPGA can deliver high performance with the levels of precision needed by scientific computing[Underwood 2004; Underwood and Hemmert 2004; Hemmert and Underwood 2005; Govindu et al. 2003; Govindu et al. 2004b; Govindu et al. 2004a; Zhuo and Prasanna 2005a; Zhuo and Prasanna 2005b; Zhuo and Prasanna 2004; Dou et al. 2005; deLorimier and DeHon 2005; Zhuo and Prasanna 2005c; Morris et al. 2006; Kindratenko and Pointer 2006; Scrofano et al. 2006]. The weakness in the majority of these studies, however, is that they do not consider either the API to deliver the performance to the application or the system architecture issues such as bandwidth and latency to the accelerator.

Of these recent works, only a handful have discussed how the performance could be incorporated with an application. The three most notable examples are a molecular dynamics application[Kindratenko and Pointer 2006; Scrofano et al. 2006], a full CG solver[Morris et al. 2006], and a traffic simulation engine[Tripp et al. 2005]. None of these efforts have considered requirements for future systems attempting to leverage FPGAs.

In addition, each of these cases focus on a fully custom data and control path. While this will ultimately be the best approach to deliver the maximum possible performance, many advocates of accelerators in general believe that the first path to general adoption is through the acceleration of common scientific computing APIs. This paper considers what will be required to make that approach work.

# 8 Conclusions

It is important to recognize the mismatch between the way FPGA researchers assess FPGA capabilities and the way applications would use the devices. Accelerator proponents

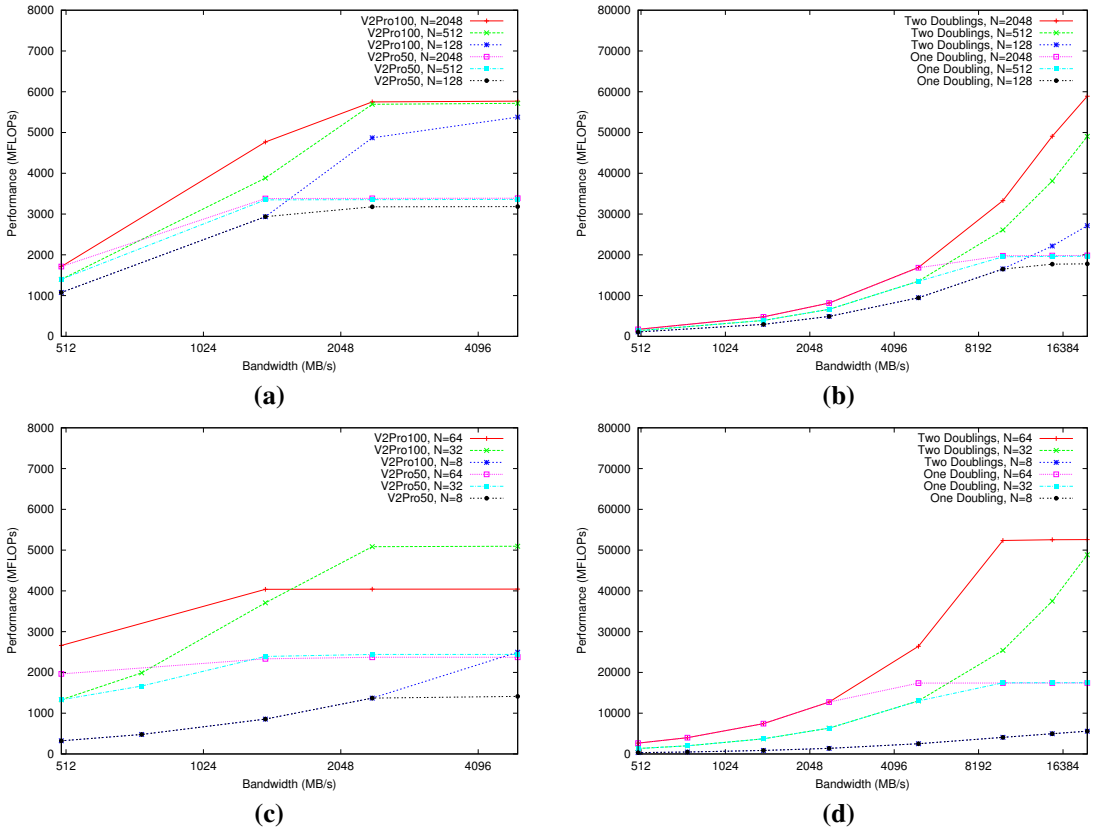


Figure 10: Simulated impact of bandwidths as FPGA capabilities scale for FFT ((**a**) and (**b**)) and DGEMM ((**c**) and (**d**))

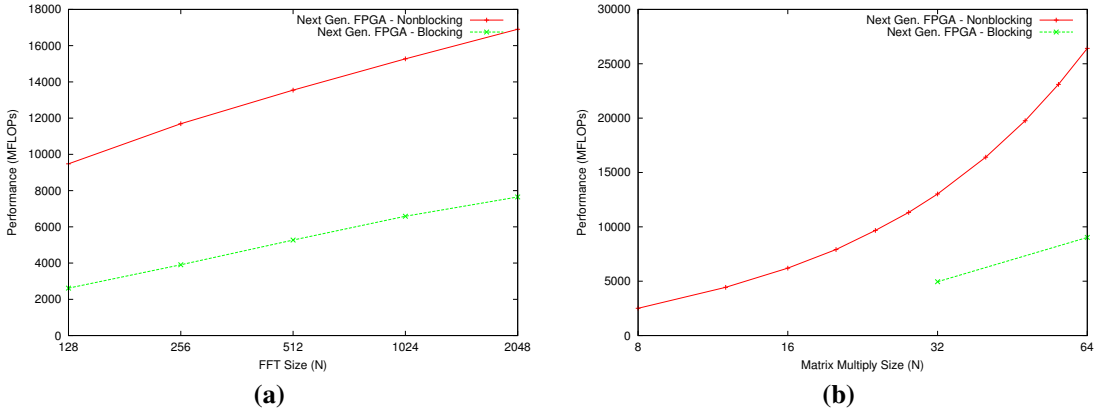


Figure 11: Simulated impact of blocking operations for **(a)** FFT and **(b)** DGEMM

like to claim that they can accelerate standard libraries, but many applications do not call these libraries in a way that can be exploited; thus, a new, non-blocking API is needed to expose the parallelism to the hardware. Indeed, the issue of API impacts performance as much or more than the issue of architectures with as much as a 3× loss in performance if the wrong API is used.

That said, architecture is critically important. The performance that the Xilinx Virtex4-FX140 should deliever will far outstrip the bandwidth that is readily available to it. With 5 GB/s/direction of bandwidth, this part (which should be available soon) would greatly outstrip the capabilities of a microprocessor. And, modern interfaces can deliver almost that level of bandwidth, but FPGA I/Os often limit the performance of those interfaces (e.g. HT). This challenge multiplies with the next generation, where those parts cannot be distinguished from current parts without delivering drastically more bandwidth.

As a final note, while these results were obtained with an FPGA focus, many of them are generally applicable to accelerator technologies. Accelerator proponents typically offer to accelerate DGEMM and FFT operations by attaching an accelerator to a commodity microprocessor and using standard BLAS calls. These results indicate that this may work in the near term for large operations, but in the longer term, the traditional blocking BLAS calls will hide the parallelism that is required. Furthermore, for many of the DGEMM and FFT operations used by scientific applications, the API is already a major barrier and the system architecture (particularly bandwidth) will be an enormous barrier in the near future. This is not meant to say that accelerators are infeasible, only that traditional library calls will not be a productive way to use them.

## References


BURGER, D., AND AUSTIN, T. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

DELORIMIER, M., AND DEHON, A. 2005. Floating point sparse matrix-vector multiply for FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*.

DOU, Y., VASSILIADIS, S., KUZMANOV, G., AND GAYDADJIEV, G. 2005. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*.

FRIGO, M., AND JOHNSON, S. G. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, vol. 3, 1381–1384.

GOVINDU, G., ZHUO, L., CHOI, S., GUNDALA, P., AND PRASANNA, V. K. 2003. Area and power performance analysis of a floating-point based application on FPGAs. In *Proceedings of the Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003)*.

GOVINDU, G., CHOI, S., PRASANNA, V. K., DAGA, V., GANGADHARPALLI, S., AND SRIDHAR, V. 2004. A high-performance and energy-efficient architecture for floating-point based lu decomposition on fpgas. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*.

GOVINDU, G., ZHUO, L., CHOI, S., GUNDALA, P., AND PRASANNA, V. K. 2004. Analysis of high-performance floating-point arithmetic on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*.

HEMMERT, K. S., AND UNDERWOOD, K. D. 2005. An analysis of the double-precision floating-point fft on fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

JANSSEN, C. Personal communications.

KINDRATENKO, V., AND POINTER, D. 2006. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

MORRIS, G. R., PRASSANA, V. K., AND ANDERSON, R. D. 2006. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

PLIMPTON, S. J., POLLOCK, R., AND STEVENS, M. 1997. Particle-mesh ewald and rRESPA for parallel molecular dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*.

PLIMPTON, S. J. 1995. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics 117*, 1–19.

SCROFANO, R., GOKHALE, M., TROUW, F., AND PRASANNA, V. K. 2006. A hardware/software approach to molecular dynamics on reconfigurable computers. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

TRIPP, J. L., MORTVEIT, H. S., HANSSON, A. A., AND GOKHALE, M. 2005. Metropolitan road traffic simulation on fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

UNDERWOOD, K. D., AND HEMMERT, K. S. 2004. 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*.

UNDERWOOD, K. D. 2004. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*.

WILLIAMSON, D. L., DRAKE, J. B., HACK, J. J., JAKOB, R., AND SWARZTRAUBER, P. N. 1992. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *J. Comput. Phys. 102*, 211–224.

ZHUO, L., AND PRASANNA, V. K. 2004. Scalable and modular algorithms for floating-point matrix multiplication on fpgas. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*.

ZHUO, L., AND PRASANNA, V. K. 2005. Design trade-offs for blas operations on reconfigurable hardware. In *Proceedings of the International Conference on Parallel Processing (ICPP)*.

ZHUO, L., AND PRASANNA, V. K. 2005. High performance linear algebra operations on reconfigurable systems. In *Proceedings of the ACM/IEEE SC2005 Conference*.

ZHUO, L., AND PRASANNA, V. K. 2005. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*.