A Practical Measure of FPGA Floating Point Acceleration for High Performance Computing

John D. Cappello
Optimal Design, Inc.
Sewell, NJ
jcappello@optimal-design.com

Dave Strenski
Cray, Inc.
Ypsilanti, Michigan
stren@cray.com

***Abstract*—A key enabler for Field Programmable Gate Arrays (FPGAs) in High Performance Computing (HPC) has been the addition of hard arithmetic cores. These "slices of DSP" dedicated to accelerated number crunching allow FPGAs to deliver more computing muscle, especially for floating point algorithms. This paper compares how an FPGA's performance in a practical HPC application measures up to its theoretical capacity. The implementation of a floating point matrix multiplication algorithm based on a 12x12 MAC (Multiply-Accumulate) array targeting the Xilinx Virtex 7 XT family is described. Several design techniques were used to ensure uninterrupted systolic operation of the array throughout execution, including a novel approach to handling heavily pipelined accumulators, as well as a scheme for overcoming the inherent inefficiencies of DDR3 memory. The result is a sustained "practical" performance range of 144-180 GFLOPS, compared to the target device's "theoretical" range of 257-290 GFLOPS.**

Keywords-FPGA; matrix multiplication; high performance computing; floating point arithmetic; multiply-accumulate; systolic array; hardware acceleration; GFLOPS; Xilinx; Virtex-7; DSP48; heavily-pipelined accumulators

I. INTRODUCTION

High performance computing (HPC) is a realm of computer science that relies on advanced, highly parallel computing systems applied specifically to assist scientists, engineers, and even financial analysts in executing complex, arithmetically-intensive algorithms for solving problems in their respective areas of study and application. Computer architectures geared toward HPC are typically comprised of an array of processing elements (PEs) configured to accelerate these complex algorithms in a manner that takes advantage of the aggregate performance benefits of parallelism.

As recently as ten years ago, a PE was simply a microprocessor operating within the limitations of the venerable von Neumann computer architecture. Today, the HPC user has three PE technologies to choose from: multi-core processors (MCPs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs). For more custom-targeted computing needs, the Application-specific IC (ASIC) or digital signal processor (DSP) devices have been utilized as a rare fourth option.

Many issues come into play when selecting a PE for a particular application, such as performance, power, cost, complexity, adaptability to a range of algorithms, and conformance to a user's computer environment. It also helps if the solution adheres to existing standards and protocols. This is where FPGAs have lagged behind. Despite three decades on the market, designing for an FPGA still requires a set of niche skills and use of non-standard development tools that typically fall outside the expertise of most HPC users. Furthermore, the complexity of an FPGA's development cycle makes pushing its theoretical performance limits challenging even to experienced developers, giving MCPs and GPUs and their standard development platforms a clear advantage in the eyes of many system architects and HPC users.

Nonetheless, there are overwhelming benefits to using FPGAs; in some cases, the development effort is worth the investment. An FPGA allows the construction of hardware architectures that are fine-tuned toward specific applications. Bioinformatic algorithms such as Smith-Waterman which are commonly used for DNA sequencing alignment match up well with an FPGA's spatial and temporal parallelism capability [16]. An FPGA's superior energy efficiency makes it a competitive choice for implementing Basic Linear Algebra Subroutines (BLAS), a key library of functions for scientific applications [6]. For molecular modeling, an FPGA's dedicated pipeline structures, low latency communication threads, and flexible algorithmic restructuring makes it a preferred platform for modeling the iterative Newtonian interactions of atoms and molecules [15]. The financial world has made huge investments in FPGA technology for High Frequency Trading (HFT) applications, specifically to take advantage of an FPGA's ability to handle electronic trade data with very low latency and minimal jitter [17].

High-end FPGA architectures are generally comprised of configurable logic blocks, embedded ram, and hard arithmetic cores arranged in a semi-regular pattern. With its sea of reconfigurable logic, the FPGA is the ultimate sandbox for digital computing, yielding unrivaled flexibility and an inherent parallelism that cannot be approached by any other technology.

As vast as this potential seems for FPGAs, there are practical limitations when it comes to crafting an actual design. This paper demonstrates these limitations by describing the implementation of a floating point matrix multiply function—a workhorse of many scientific algorithms—in an architecture designed to take full advantage of an FPGA's arithmetic computing power. The matrix multiply is a standard linear algebra function that greatly benefits from parallelism, where tremendous performance gains can be had when accelerated

within an HPC platform. It's also used as a standard benchmark for evaluating the performance of HPC machines.

The flow of this paper begins with a look at related work on matrix multiplication implementations in FPGAs, followed by an analysis of an FPGA's theoretical limits for both optimal usage of resources and specifically for matrix multiplication. Next, the mechanism for mapping the matrix multiply algorithm onto FPGA fabric for this implementation is described, with a look at several issues that were addressed for overcoming I/O bound performance. Several design techniques that were employed to ensure uninterrupted systolic operation of the MAC array are described, including the handling of heavily-pipelined accumulators, devising an optimal schedule for feeding matrix data to the MAC array, and maximizing DDR3 memory efficiency. The paper concludes with the floorplanning strategy behind packing as many MACs onto the FPGA die as possible while meeting timing closure, along with potential opportunities for squeezing more performance out of the design.

II. Related work

Numerous architectures for implementing matrix multiplication onto FPGAs have been described in recent years [3], [5], [6], [7], [11]. The motivations behind these efforts have varied, ranging from creating an efficient, scalable, and high performing architecture, to evaluating how an FPGA compares to CPUs, MCPs, and GPUs for HPC applications.

With the ubiquitous matrix multiply algorithm being used for this demonstration, many of the same tradeoffs and issues discussed in previous work came into play in this design, such as I/O bound performance, matrix blocking, and the importance of data re-use. But there were architectural differences. While others adopted a linear array of PEs with data streaming to and from the array through a single endpoint PE, this implementation is based on a 2D array of PEs, with data distributed into ram banks feeding the array's rows and columns. In other work, a host processor interacted directly with the FPGA throughout execution, downloading and uploading data directly with the FPGA's internal PE array. In this architecture, the FPGA executes the algorithm autonomously while uploading and downloading data with tightly-coupled DDR3 memory.

III. Pushing the Theoretical Performance Envelope

A key aspect to this demonstration is the use of floating point arithmetic, a computing method required for many scientific applications where dynamic range and accuracy are critical. Not long ago, FPGAs were grossly inefficient with these types of calculations; the amount of logic needed to realize multiplication was too taxing on fabric resources.

Times have changed. One of the greatest enablers of FPGAs for HPC today is the proliferation of hard multiplier cores now available in many high performance FPGA families. This has boosted an FPGA's capabilities considerably in performing floating point computations.

The hard multiplier core offered by the Xilinx Virtex 7 Series family—the highest performing Xilinx family on the market today—is the DSP48. The DSP48 contains a 25x18 multiplier, a 48-bit accumulator, and an assortment of flexible options for implementing a number of arithmetic operations. It's also the perfect digital hub for creating a floating point MAC unit, a key processing element for matrix multiplication.

The device targeted for this implementation was the Virtex-7 XC7VX690T, which contains as many DSP48s (3,600) as any FPGA offered by Xilinx. The number of DSP48s required to realize a single MAC unit—critical to fitting the maximum number of MACs on a single die—depends on the desired precision and implementation scheme. The LogiCORE IP Floating-Point Operator tool [9] from the Xilinx ISE development platform can create an assortment of functional floating-point operations with various speed-area options, all of which impact the number of DSP48s realized. For instance, a double precision floating point multiplier configured for high speed and maximum usage of hard multipliers within the XC7VX690T requires 11 DSP48s.

According to theoretical analysis in [1], the Virtex 7 Series FPGAs can perform double precision floating point arithmetic about 4.2 times faster than a 16-core MCP. Based on a scheme that allocates the available fabric resources with various combinations of arithmetic operator types configured in an assortment of realization schemes, the theoretical peak performance of the XC7VX690T was measured at 290 GFLOPS.

This paper extends this theoretical research by implementing as large of a MAC array as practical in the same device. With one multiplier and one adder needed per MAC, matrix multiplication requires a homogenous set of MAC units for repeatability, consistency, and efficiency. For this implementation, each adder and multiplier was configured for maximum usage of DSP48s to achieve the highest clock frequency possible. (Later we'll see that "highest frequency" doesn't necessarily lead to highest overall performance.)

To calculate the theoretical performance limit of the matrix multiply application for the XC7VX690T, the maximum number of MACs that can fit is determined, along with a target operating frequency. As mentioned above, each multiplier utilizes 11 DSP48s. Similarly, a floating point adder requires three DSP48s. Together, these operators make up a floating point MAC unit encompassing 14 DSP48s. Using simple math, the number of MACs that could theoretically be accommodated inside a XC7VX690T is 257 (3600 DSP48s / 14 per MAC). The target clock rate depends on the performance specifications. For XC7VX690T in a -3 (highest) speed grade [10], it was judged that a clock frequency of 500 MHz could be attainable with reasonable effort. Hence, the overall theoretical peak performance of a floating point matrix multiply algorithm implemented in the XC7VX690T-3 is calculated to be 257 GFLOPS (2 FLOPs per MAC x 257 MACs x 500MHz), about 11% less than the optimal combination of adder/multiplier types calculated in [1].

IV. Projecting Matrix Multiplication onto an FPGA

The following example demonstrates how the matrix multiply architecture was devised for this implementation.

In Figure 1, two 4x4 matrices denoted as **A** and **B** are multiplied together to form a set of results in the form of a **C** matrix. Each **C** matrix term is the result of the dot product calculation of an **A** matrix row vector with a **B** matrix column vector. Since a dot product calculation is nothing more than a sequence of multiplication results summed together, each of the 16 dot products required to produce the 4x4 **C** matrix can be parallelized by mapping them to dedicated arithmetic blocks configured as MAC units.

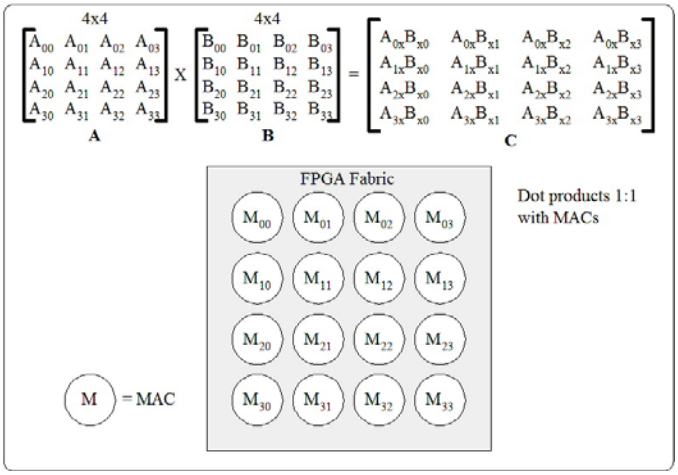

Figure 1. Mapping of matrix multiply to an FPGA MAC array

In the previous section, it was calculated that a maximum of 257 MACs could theoretically fit inside a XC7VX690T. This would be enough to contain a 16x16 MAC array. From a practical standpoint, however, it would be extremely difficult to get an actual implementation close to this theoretical limit. There are several reasons why, and each plays a role in the final architecture. It's a process of scaling down what is theoretically possible in an FPGA to what is practical.

V. Overcoming I/O Bound Performance

Fabrication technology strongly influences the maximum clock frequency and the total number of MACs that could fit on a device, two major factors in determining overall performance of the matrix multiply algorithm. But the greatest *architectural* impact is I/O.

Targeting this algorithm for HPC, large amounts of data would be expected to be transferred to and from an FPGA's I/O pins continuously throughout execution. An I/O link's inability to keep pace with the core processing power of the FPGA is a major deterrent to aggregate performance. Because of the nature of how I/O technology will always lag behind on-chip processing from a raw processing power standpoint, a matrix multiplication implementation will become *I/O bound* unless this interaction can be handled efficiently enough to be classified as "background traffic."

For this implementation, the I/O bound situation as related to a host interface was eliminated by adding local bulk memory to the FPGA in the form of two DDR3 modules: a 4GB DIMM for storing the input **A** and **B** matrices, and a 2GB DIMM for holding the **C** matrix results. These memories essentially decouple a host's interaction from impacting the FPGA's overall performance.

The MAC array's performance can still become I/O bound with regard to the DDR3 links. Three architectural enhancements were employed to alleviate this: *blocking*, *data re-use* and *local MAC cache*. These will be described in the following sections.

VI. Matrix Multiplication Architecture

The core of the matrix multiplication architecture is a 12x12 systolic MAC array, shown in Figure 2. **A**/**B** matrix data are uploaded from DDR3 memory and distributed into ram banks that act as gateways into the array's rows and columns. A second DDR3 memory link acts as a depository for the array's **C** matrix data results.

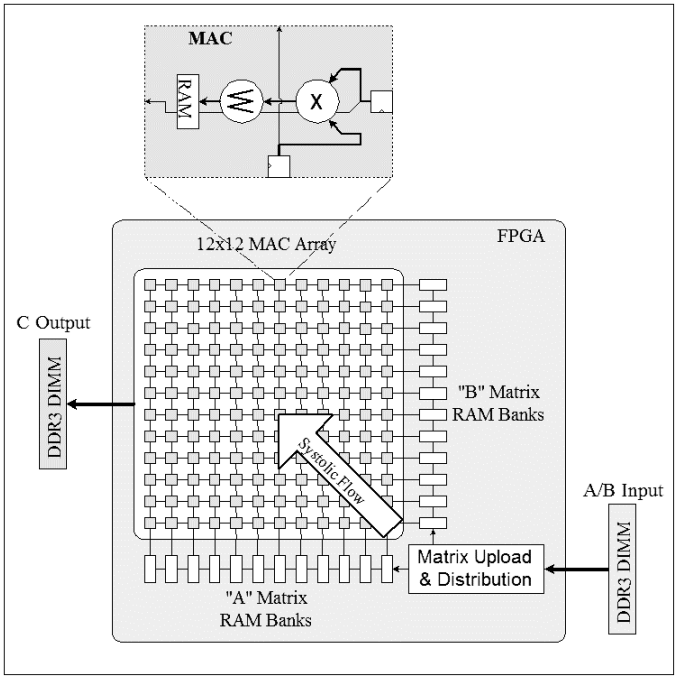

Figure 2. Top view of the matrix multiplication architecture

The key to the array's systolic operation is the manner in which the **A**/**B** matrix data is distributed and sequenced, which ultimately determines how the array is able to perform 144 dot product calculations in parallel throughout its execution.

Data is staggered into the array and propagated through each linear chain so that it can be re-used by each MAC in its path. Figure 3 shows how the systolic operation would begin, with each cycle time *t{i}* representing a systolic beat. The array becomes saturated shortly after startup and remains so until ramping down toward the end of its execution. Because of the long duration of performing a matrix multiplication for very large arrays, the startup and ramp down times are negligible when factored into overall performance.

Once a MAC has completed a dot product calculation for a given set of vector data, it will either deposit the result into its local cache as a "partial sum" to be accumulated later for

subsequent dot products, or it will output the result as a "final sum."

Several additional design techniques were needed to ensure that the MAC array operates in systolic fashion and without interruption throughout its execution. These are discussed in the next few sections.

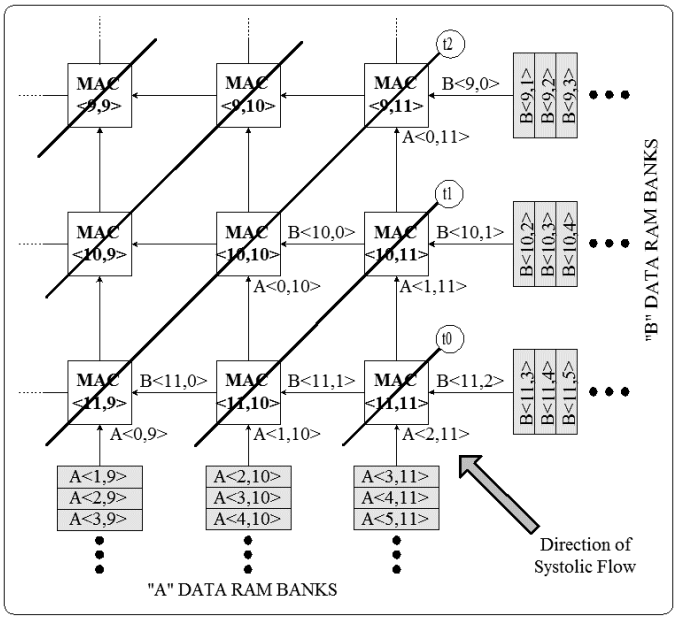

Figure 3. The start of the MAC array's systolic operation

VII. Handling Heavily-Pipelined Accumulators

The maximum frequency that an FPGA design can be clocked is by definition limited by that design's slowest combinatorial path. A designer can raise the headroom on clock frequency, and in most cases, on overall performance, by adding pipeline registers strategically throughout the various hardware structures of the architecture.

Complex functions such as multipliers and adders typically require many layers of combinatorial logic. The more complicated the function, the more pipeline stages needed in order to achieve a certain level of performance. [For a component replicated many times in a design such as in an array, additional pipelining could actually be a detriment to performance because of the increased area; this is addressed in section XI, "Squeezing more performance."]

With pipelining comes latency. A double precision floating point multiplier configured for maximum clock rate (as generated by the Xilinx tools while targeting a Virtex-7 device) contains 16 pipeline stages, corresponding to a latency of 16 cycles. The ramifications of interfacing with a heavily pipelined multiplier in a parallel processing environment is minimal in that its inputs and outputs could still be streamed continuously at its clock rate; the designer needs only to account for the fixed latency as to when to expect valid data on its output.

On the other hand, a heavily pipelined accumulator presents a much greater challenge and has an acute impact on the overall architecture. Unlike the multiplier, data can't just be streamed into a heavily pipelined accumulator because each sum it produces needs to be fed back immediately into one of its inputs to satisfy the accumulate operation. It's a classic dilemma for system designers and has been thoroughly investigated [2], [4], [13], [14]. The simplest option, sufficient for most low performance applications, is to simply delay the input stream as each accumulation operation runs its course. But for higher performance, this solution would cripple data throughput. Another option is to replace the accumulator with an adder tree fed by an array of multipliers so that the dot product multiplications are no longer accumulated at a single point, eliminating the feedback requirement. But adder trees take up huge amounts of resources that drastically reduce the number of MACs that could fit in the FPGA, which in turn shrinks overall performance. Other more complex solutions have been proposed [2], [4], [13], [14].

For this implementation, a special technique was adopted that didn't require the input stream to be delayed, nor did it require additional arithmetic resources. It's called *accumulator time-sharing*. Instead of sending a continuous stream of multiplier results into an accumulator one set at a time, where a "set" corresponds to a single dot product calculation, multiplier results from multiple sets are *interleaved* in a manner which allows the accumulator to be time-shared. The interleaving eliminates the need to use an adder's result immediately, at least from one cycle to the next. Instead, the adder results are stored in local cache for subsequent feedback into the accumulator during a later cycle.

The number of sets configured to time-share the accumulator must be greater than the latency of the accumulator, plus additional cycles to account for the path of partial sums as they are propagated back into the accumulator. For this application, 25 shared sets was determined to be more than enough to support a heavily pipelined, 14-stage adder while encompassing all latencies associated with the propagation chain.

VIII. Block Matrix Rolling Schedule

A "Block Matrix Rolling Schedule" defines the pre-determined sequence of **A** and **B** block matrices that are uploaded from DDR3 memory and consumed by the MAC array throughout the matrix multiply execution. This section describes how the schedule was derived, with the primary goal of keeping the systolic MAC array running without interruption.

Performing a matrix multiplication of two 12x12 arrays isn't an operation that would provide much benefit if accelerated in hardware. But performing a matrix multiplication of two 12,000x12,000 arrays would. To handle very large arrays, the FPGA processes smaller blocks of the arrays in a technique called *matrix blocking*. To demonstrate how matrix blocking works, consider a 12x12 matrix multiplication shown in Figure 4.

The upper left term {0,0} of the **C** matrix is calculated by performing a dot product on the first row vector of the **A** matrix with the first column vector of the **B** matrix.

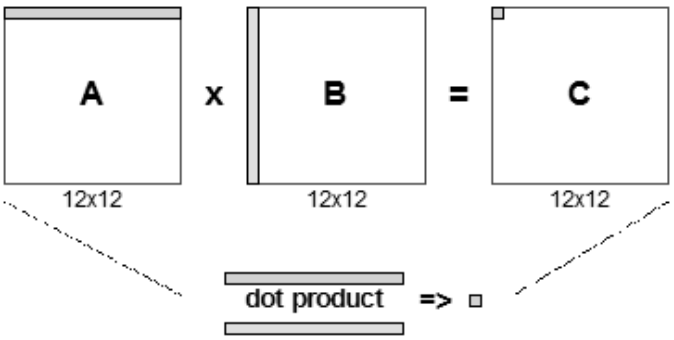

Figure 4. Sample dot product for 12x12 matrix multiply

Now suppose this 12x12 matrix multiplication is to be performed on a 4x4 MAC array. One strategy is to split the 12x12 arrays into sixteen 4x4 blocks, then perform a series of 4x4 matrix multiplies directly on the 4x4 MAC array. Calculating the {0,0} **C** term would then require four 4x4 matrix multiplies, as shown in Figure 5. After each matrix multiply, a partial sum is generated each time the **A** block rolls right in conjunction with the **B** block rolling down. The FPGA would maintain a running set of partial sums until a final **C** term result was reached.

However, this plan breaks down performance-wise if applied literally for very large arrays. The I/O demand for uploading **A**/**B** matrix data and feeding it into the MAC array would far exceed the capable bandwidth of the DDR3 link supplying the data, causing the array to stall periodically and degrading overall performance.

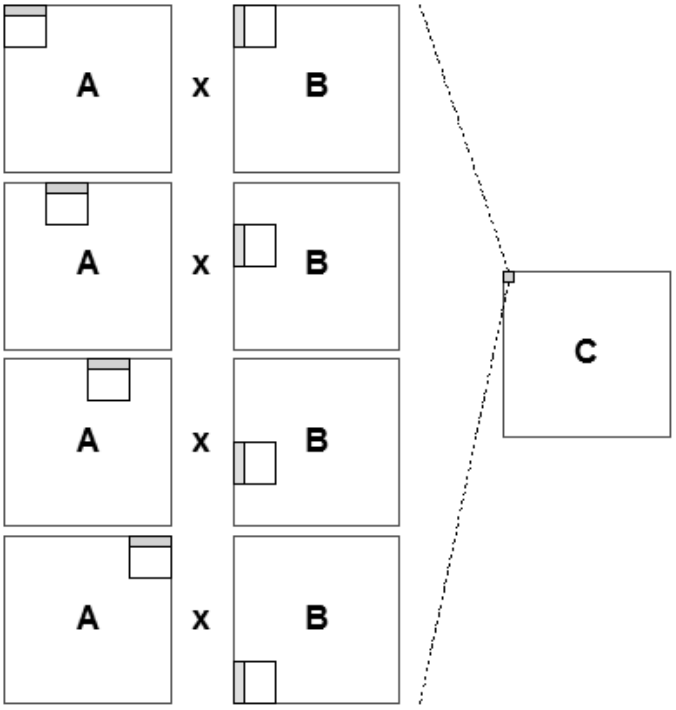

Figure 5. Matrix Blocking: 12x12 array split into sixteen 4x4 blocks

To see this, consider the matrix multiplication of two 12,000x12,000 arrays on a 12x12 MAC array. In this scenario, each of the large arrays is processed in 12x12 blocks to match the MAC array. Using the matrix blocking procedure demonstrated above, the upper left **C** array term {0,0} would be calculated from a series of one thousand 12x12 matrix multiplies performed similar to the flow in Figure 5.

This sequence is functionally sound. However, if the **A**/**B** "consumption rate"—i.e. the rate that data is consumed by the MAC array—is compared to practical DDR3 link rates, it can be shown that the MAC array would have to stall periodically waiting for new **A**/**B** data to be uploaded from off-chip memory.

Consider that a standard DDR3-1600 64-bit DIMM module operates at a maximum line rate of 102 Gbps. Because of the inefficiencies of DDR3 technology related to inherent latencies [8], a 50% efficiency rate is deemed a reasonable goal for this link. This would pin the maximum *allowed* rate of uploading **A**/**B** matrix data at 51 Gbps.

The **A**/**B** consumption rate is calculated by taking the total number of data bits consumed by the MAC array over a specific time interval. For the block matrix rolling schedule described above, since new **A** and **B** matrix blocks are required for every 12x12 matrix multiplication, the time of a single matrix multiply, 24 nanoseconds (12 systolic clock cycles @ 500 MHz), is the interval used for consumption calculation. Two double precision floating point matrix blocks contain 18,432 bits (12x12 array x 2 x 64 bits). The corresponding **A**/**B** consumption rate is therefore *768 Gbps* (18,432 / 24nsec), a rate which far exceeds the 51 Gbps limit set for the DDR3 link.

If the interleaving method required to handle the heavily pipelined accumulators is taken into account, the consumption rate can be reduced significantly. A new rolling schedule is introduced below which interleaves 25 dot products at one time while time-sharing the accumulator. Instead of the **A** and **B** blocks rolling in conjunction with each other, the **A** block *is kept static* while the **B** block is rolled horizontally 25 times, as shown in Figure 6.

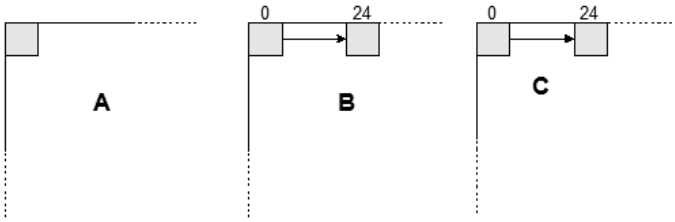

Figure 6. Static "A" block with rolling "B" block

Keeping the **A** block static relieves some of the I/O demand; instead of 50 blocks needed for 25 matrix multiplies, only 26 (1**A** + 25**B**) are needed. However, this consumption rate (384 Gbps) is still too excessive, though it is important to note that the introduction of *data re-use* made a significant contribution here in reducing the I/O demand by 50%. But clearly more data re-use is needed.

It turned out that the strategy for transferring *only* the final results from the array off chip to its own output DDR3 memory helped reduce the **A**/**B** consumption rate to a satisfactory level. For efficiency, it was decided early on that *partial* **C** results should not be cached to external DDR3 memory, and that only the *final* results are to be transferred off chip once they have been obtained. This decision reduced the overall design complexity as well as eliminated the write path from the output DDR3 link to the MAC array–greatly improving overall layout and performance margins.

This strategy also required each MAC to maintain a set of partial sums locally until final sums are reached. How many sums should be stored? The fabric resources would determine this. The fundamental size of an embedded block ram (known as *BRAM*) inside the Virtex 7 Series family is 512x32. Considering that the XC7VX690T has a capacity of 2,940 BRAMs, it was deemed reasonable to allocate two BRAMs per MAC to hold up to 512 64-bit sums locally. The 512x64 cache ram would then be able to hold partial sums for *20 groups* of 25 time-shared dot products at one time.

With this allocation, a final rolling schedule can be described. This schedule is composed of five concentric loops. The first three loops are shown graphically in Figure 7.

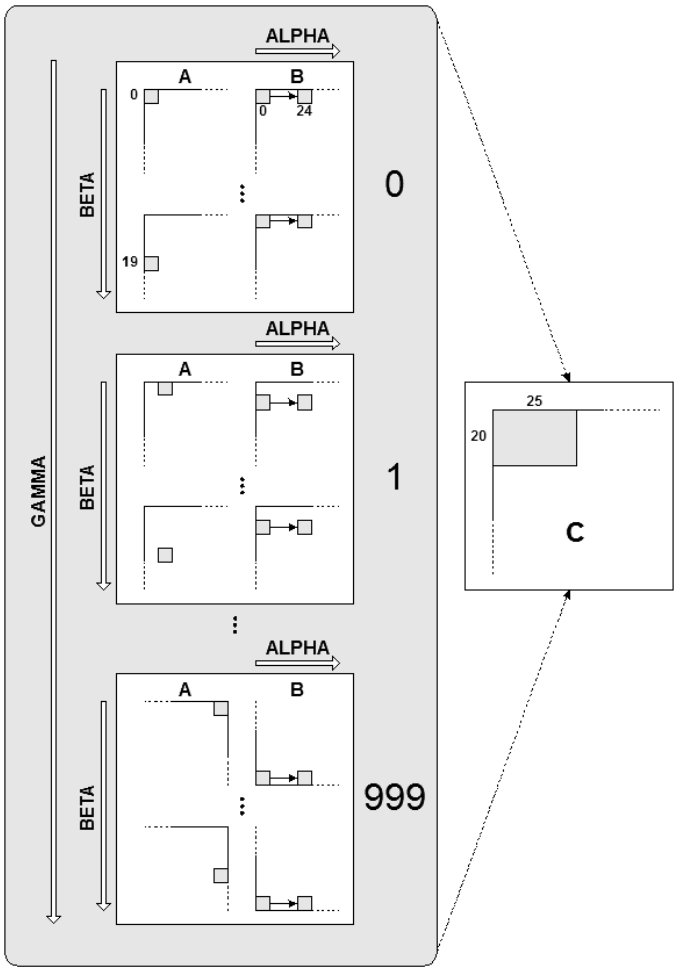

Figure 7. First three loops of block matrix rolling schedule

The first loop, *ALPHA,* corresponds to the time-sharing of 25 consecutive dot products, where a single **A** block is multiplied across 25 **B** blocks. The second loop, *BETA,* involves the local cache parameters described above. Since 20 sets of 25 dot products are to be cached in each MAC at one time, *BETA* essentially repeats *ALPHA* 20 times for 20 different **A** blocks, but for the same set of 25 **B** blocks.

Each *BETA* loop represents a snapshot of the blocks of data that can be cached inside the FPGA at one time. Hence, this period is the basis for calculating the maximum **A**/**B** consumption rate for this rolling schedule. With 45 blocks processed over 500 matrix multiplies (20 **A** x 25 **B**), the calculated consumption rate is *34.5 Gbps* (45x12x12x64 / 500 x 24 nsec), well within the stated DDR3 link goal of 51 Gbps. This means that for this block matrix rolling schedule, as long as the DDR3 link for uploading **A**/**B** matrix data can run at an efficiency of at least 50%, there should be plenty of margin for the systolic array to churn uninterrupted.

IX. Maximizing DDR3 Memory Efficiency

DDR3 memories are streamlined to operate at a minimum burst length of eight data words. Any access to the memory that isn't a multiple of eight leads to wasted cycles and link inefficiency. This would typically be a performance obstacle for random memory accesses of varying sizes. But for this implementation, the DDR3 bursts are continuous and long enough such that any inefficiency is negligible.

A more common—and often underestimated—detriment to DDR3 efficiency are the inherent latencies associated with *activation* and *precharge* times as memory rows are accessed. These latencies are relatively slow compared to the link's maximum burst rate and are often the culprits behind poor DDR3 performance. For many applications, a designer's hands are tied regarding the ability to combat these DDR3 inefficiencies because of lack of control over the activity across the DDR3 link. For this application however, the designer has a priori knowledge of how data will be uploaded from DDR3 memory. Thus, **A**/**B** matrix data can be stored in DDR3 memory in such a way that counteracts the latency effects.

Consider the graphical representation of a typical 4Gb DDR3 address configuration in Figure 8. This particular device contains eight banks of RAM, with each bank oriented with a 16-bit row address (i.e. 64K rows) and a 10-bit column address (i.e. 1024 columns).

Since DDR3 memory accesses are burst-oriented, the startup latency due to row activation applies only to the output of the first data; all subsequent data of the burst will follow at line rate. As long as the row address remains constant, no additional latencies will be incurred. Figure 8 shows that up to seven contiguous 12x12 matrix blocks can fit within a single DDR3 row. This allocation would ensure that every matrix block can be retrieved in its entirety at the DDR3 line rate.

To get the **A**/**B** data stored in this manner isn't trivial. Matrix data is commonly stored in "row major order" or "column major order," depending on the software technique used for generating and storing the data. This means that for a 12,000x12,000 array, each row (or column) of 12,000 elements

is stored linearly. When the host downloads these arrays into the FPGA, the FPGA would have to re-map the data on-the-fly into specific DDR3 addresses to ensure that the 12x12 matrix blocks are deposited within the same DDR3 row to guarantee that each stored block can be completely uploaded from DDR3 at line rate. This re-mapping should be transparent during host download without impacting overall performance.

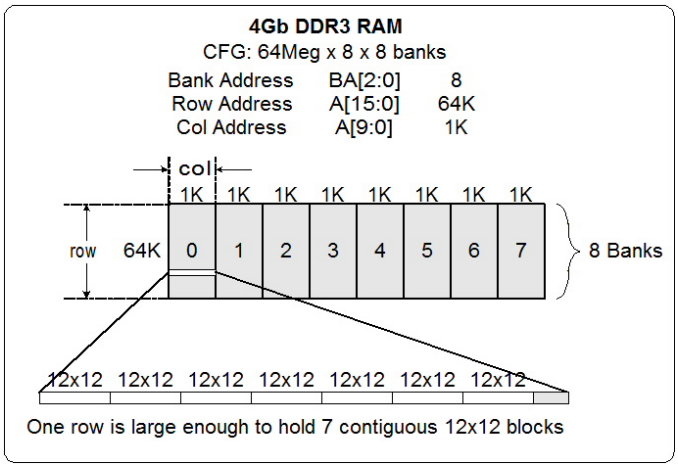

Figure 8. Typical DDR3 address configuration and block matrix allocation

X. Floorplanning and Timing Closure

One of the goals of this project was to pack as many MACs onto the die as possible. In general, for a given clock frequency, more MACs means higher performance. At 500 MHz and two floating point operators per MAC, each MAC effectively contributes 1 GFLOPS of performance to the overall operation.

Ironically, an FPGA's superior flexibility in the placement and routing of customizable logic makes it that much more difficult to estimate how fast a design could possibly perform, which is why developers will often do preliminary routes. But a device's switching specifications can be used as a baseline.

According to the Xilinx Virtex-7 datasheet [10], configurable logic blocks (CLBs) in a –3 speed grade can be clocked at a max rate of 685 MHz; a fully pipelined DSP48 can reach 617 MHz; BRAMs top out at 601 MHz. After examining these specifications and going through trial iterations using much smaller MAC arrays, it was deemed that a 12x12 MAC array clocked at 500 MHz would be attainable with reasonable effort.

Two factors that directly impact the number of MACs that can fit on the die are the layout of the fabric resources and the placement requirements for the arithmetic cores. Figure 9 shows the general arrangement of functional blocks on the XC7VX690T die. The fabric is comprised of a very large array of logic slices which are separated by columns of DSP48s and BRAMs. Some of the layout asymmetries of this device–not obvious from the figure–should be noted. For instance, there are three rectangular voids along the right side of the die, each dedicated to a hard core implementation of a PCIe Gen3x8 link. Since routes between MACs can't travel over these voids, this is an area vulnerable to difficulties in getting adjacent MACs to communicate at high frequency, which in turn hinders the efficiency of packing the MACs onto the die. Another asymmetry is the ratio of DSP48 columns (18) to BRAM columns (15). Each MAC relies on both these resources, so the fact that this ratio isn't 1:1 plays a minor role in limiting the number of MAC columns that can be mapped across the die.

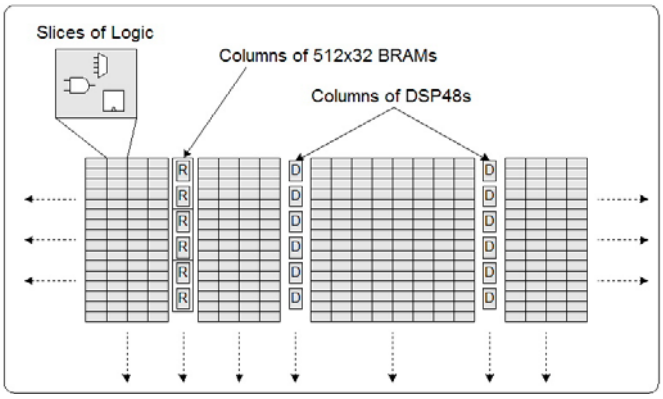

Figure 9. Virtex-7 XC7VX690T Layout

For the most part, the number of MACs per column was strongly determined by the layout requirements of the multiplier (11 DSP48s) and adder (3 DSP48s) cores created by the Xilinx Floating Point Operator tool. At 200 DSP48s per column, simple math dictates that at most 14 MACs (200 / 14) can be squeezed vertically within the locale of each DSP48 column; hence the choice of the semi-conservative 12x12 MAC array.

A design this challenging often needs a sound floorplanning strategy to meet timing closure. For this implementation, the timing results improved each time a new row of MACs was locked down to a specific area on the die. Timing closure was reached once all MACs, ram banks, and the two DDR3 controllers were constrained to specific areas of the die.

XI. Squeezing more Performance

One of the key benchmarks for gauging how well a design takes advantage of an FPGA's available assets is resource utilization. For this implementation, 56% of DSP48s and 43% of BRAM were used. Given the success of reaching timing closure at 500 MHz for the 12x12 MAC array, these numbers hint that the XC7VX690T was underutilized to a certain extent, and that there is enough slack for performance improvement. For instance, it's feasible that one more row and column of MACs can be added to the array while maintaining the same clock rate, boosting the performance to 169 GFLOPS.

A BRAM's datasheet timing for a XC7VX690T-3 drops to 529 MHz if cascaded to create larger memories. This was the case inside each MAC for this implementation, making BRAMs the clear performance bottleneck. To gain more performance headroom so that more MACs can be added or to raise the clock frequency, a potential improvement would be to eliminate the cascade configuration of the BRAMs.

Another opportunity is the re-generation of the multiplier and adder cores to reduce each MAC's area. These cores were

originally created for highest clock rate possible, which required maximum pipelining. Since added pipelining requires more area [12], it's possible that more MACs could be added if less pipelining were used. Clock frequency would presumably drop to support lesser-pipelined cores, but this solution might still yield greater overall performance. With 18 columns of DSP48s and 15 columns of BRAM available in a XC7VX690T, a 15x15 array could feasibly be accommodated. If clock frequency were dropped to, say, 400 MHz to compensate for the smaller cores, overall performance would jump to 180 GFLOPS, a 25% increase over the original design parameters despite the 20% drop in frequency. This type of tradeoff in determining the optimal combination of clock frequency with PE density is a critical decision for any parallel processing environment, especially when factors such as cost and power consumption are considered.

Yet another area of optimization pertaining to the arithmetic cores is a change in the precision, rounding and accuracy requirements. Going from floating point to fixed, for example, would have a significant postive impact on the number of operations that could be performed per second.

XII. Conclusion

The theoretical performance capacity of the Xilinx Virtex-7 XC7VX690T FPGA for floating point applications was calculated to be in the range of 257-290 GFLOPS. A floating point matrix multiply algorithm was implemented in the same device to gauge its practical performance limits. Based on a 12x12 MAC array, two DDR3 controllers, and several architectural features designed to ensure uninterrupted systolic operation of the array, the design reached timing closure at 500 MHz for an overall sustained performance of 144 GFLOPS. Device utilization statistics and built-in slack indicate that this design could be improved to extend the practical performance range to as high as 180 GFLOPS, allowing the FPGA to reach 50-70% of its theoretical floating point capacity.

References

[1] D. Strenski, C. Kulkami, J. Cappello, P. Sundararajan, "Latest FPGAs Show Big Gains in Floating Point Performance," *HPCwire.com*, April 2012.

[2] S. Sun, J. Zambreno, "A Floating-point Accumulator for FPGA-based High Performance Computing Applications," *Field-Programmable Technology*, pp. 493-499, December 2009.

[3] Y. Dou, S. Vassiliadis, G.K. Kuzmanov, G.N. Gaydadjiev, "64-bit Floating-Point FPGA Matrix Multiplication," in *FPGA '05: Proceedings the 2005 ACM/SIGDA 13th international symposium on Field-programamble gate arrays*. New York, NY, USA: ACM, 2005, pp. 86-95.

[4] R. Scrofano, L. Zhuo, V. Prasanna, "Area-Efficient Arithmetic Expression Evaluation Using Deeply Pipelined Floating-Point Cores," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, Feb. 2008.

[5] G. Kuzmanov, W.M. van Oijen, "Floating-Point Matrix Multiplication in a Polymorphic Processor," *ICFPT 2007*, pp. 249-252, 2007.

[6] S. Kestur, J.D. Davis, O. Williams, "BLAS Comparison on FPGA, CPU and GPU," *IEEE Computer Society Symposium onVLSI*, July 2010.

[7] D.H. Jones, A. Powell, C. Bouganis, P.Y.K. Cheung, "GPU versus FPGA for high productivity computing, " *FPL 2010*, IEEE Computer Society, 2010, pp. 119-124

[8] "DDR3 SDRAM Specification," JEDEC Solid State Technology Association, July 2010.

[9] "LogiCORE IP Floating Point Operator v5.0, DS335" Xilinx Corp., March 2011.

[10] "Virtex-7 FPGAs Data Sheet: DC and Switching Characteristics, DS183," Xilinx Corp., March 2011.

[11] V.B.Y. Kumar, S. Joshi, S.B. Patkar, H. Narayanan, "FPGA Based High Performance Double-Precision Matrix Multiplication," in *VLSID 2009: Proceedings of the 2009 22nd International Conference on VLSI Design*, Washington, DC, USA, pp. 341-346. IEEE Computer Society, Los Alamitos (2009)

[12] G. Govindu, R. Scrofano, V.K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," in *Proc. of International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2005.

[13] L. Zhuo, G.R. Morris, V.K. Prasanna, "High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377-1392, 2007.

[14] M. Huang, D. Andrews, "Modular Design of Fully-pipelined Accumulators," *International Conference on Field-Programmable Technology (FTP)*, pp. 118-125, 2010

[15] B. Sukhwani, M. Chiu, Md. A. Khan, M.C. Herbordt, “Effective Floating Point Applications on FPGAs: Examples from Molecular Modeling,” HPEC 2009

[16] J. Allred, J. Coyne, W. Lynch, V. Natoli, J. Grecco, J. Morrissette, “Smith-Waterman Implementation of a FSB-FPGA module using the Intel Accelerator Abstraction Layer,” IPDPS, pages 1-4, IEEE, 2009

[17] J.W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, K. Vissers, “A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT),” HOTI, pages 9-16, 2012