FPGA-Based, Floating-Point Reduction Operations

MICHAEL R. BODNAR\textsuperscript{1}, JAMES P. DURBANO\textsuperscript{2}, JOHN R. HUMPHREY\textsuperscript{2}, PETERSEN F. CURT\textsuperscript{2}, DENNIS W. PRATHER\textsuperscript{1}

\textsuperscript{1}Electrical and Computer Engineering Department
University of Delaware
140 Evans Hall
Newark, DE 19716 USA

\textsuperscript{2}Accelerated Computing Division
EM Photonics, Inc.
51 E. Main St. Suite 203
Newark, DE 19711 USA

Abstract: - Floating-point reduction operations are a vital part of scientific computational kernels, such as vector dot-products, discrete cosine transforms (DCT), and matrix-matrix multiplications. As FPGAs continue to gain popularity in custom and embedded computing platforms, implementations of these applications in such platforms are desirable. Due to the inherently deep pipelines of high-performance floating-point cores in FPGAs, reduction circuits require special feedback and buffering schemes in order to realize full throughput. In this paper, we present our floating-point reduction architecture, clocked at more than 150 MHz targeting a Xilinx Virtex2 8000-4 FPGA.

Key-Words: - FPGA, floating-point, accumulation, reduction operations

1 Introduction
Floating-point reduction operations, including matrix-vector-multiply (MVM), vector dot-products, and the discrete cosine transform (DCT), are required as part of many scientific algorithms. These computational kernels find themselves in numerous applications, such as automatic target recognition, large-object scattering, antenna design, encryption, and long-range imaging. Many of these algorithms are being ported to custom hardware, such as field-programmable gate arrays (FPGAs), in order to achieve much greater computational power. Thus, it is clear that a high-performance, floating-point accumulation unit is necessary.

However, the difficulty in designing this type of circuit in hardware results from the deep pipelines of floating-point arithmetic units, which is needed in order to attain high performance. The long delay of a deep pipeline necessitates special handling via feedback circuits, which is further complicated by a continuous input data stream. To this end, we have developed a high-performance accumulation circuit.

In this paper, we present the key aspects of our accumulation architecture, namely storing and scheduling of the intermediate results, which allows for utilization of the underlying floating-point core approaching 100%. Our design, which is a natural evolution of previous work \cite{1, 2}, maintains buffers for partial result storage that utilize significantly less embedded memory resources than other designs, while maintaining fixed size and speed characteristics, regardless of input stream length.

We have demonstrated the power of our accumulation architectures in a matrix-matrix-multiply application in a Virtex-II 8000 FPGA clocked at over 150 MHz. Coupled with efficient DRAM controllers, careful on-chip caching techniques, and fully pipelined arithmetic units, we were able to achieve a throughput of 27.8 GFLOPs.

2 Problem Formulation

2.1 Reduction Idea
The simplest reduction circuit implementation would incorporate a single-stage floating-point unit, which could perform the reductions with no pipeline latency. However, it is widely known that the performance of non-pipelined floating-point units in FPGAs is simply too low for such designs to be practical. Although more complex methods,
requiring multiple floating-point units and special delay circuits, have also been proposed, the resource requirements of a single, deeply pipelined floating-point core alone often result in area-intensive designs [3]. The most desirable implementation requires the use of a single, pipelined floating-point unit, as described in [1, 2]. However, designing such a unit is not trivial due to the potential data hazards that arise in scheduling (pipelining) the circuit and storing intermediate results.

The input stream can be reduced, at most, once every other cycle. To visualize this, consider a constant stream of floating-point data entering a pair of input buffers in the accumulation circuit at a rate of one data item per cycle (Figure 1).

Floating-point data enters the accumulator unit once per cycle. The data is buffered and then reduced (in this case, added) every other cycle with a continuous input stream. Data exiting the computational pipe is either written to external memory (i.e., the stream has been fully reduced) or it is stored in buffers to await further reduction.

Further reduction of the partial sums can then be interleaved amongst the input data, thus taking advantage of the unused pipe stages (Figure 2). Multiple streams are reduced similarly, requiring only a small amount of additional logic to uniquely identify a given stream (Figure 3).

As stated previously, this design is a natural evolution of previous work. Zhou et al., in particular, has presented 2 different accumulation architectures [1, 2]. The first was limited to a continuous input stream, and required different implementations for different stream lengths. The second, while overcoming the shortcomings of the first, required the use of a great deal of embedded memory resources (~70 Kbits), which is undesirable for many applications. A more recent design [4] requires minimal memory resources but necessitates increased latency in multi-stream reductions, as the accumulator circuit must be emptied before the next stream can be reduced.

In contrast, our implementation uses minimal logic and memory resources, while maintaining the versatility and flexibility of non-continuous input streams, multi-stream reductions, and pipeline utilization approaching 100%.

The remainder of this paper will cover the architectural and implementation details, as well as an application section in which our design was verified in a matrix-matrix multiplier.

3 Problem Solution

Our design is based on the use of a single floating-point adder that is coupled with a buffering system and scheduler that monitors the buffers and provides data inputs to the adder (Figure 4).

In this section, we briefly describe three aspects of our accumulation architecture: scheduling partial results, storing partial results in temporary buffers, and performing multi-stream reductions. It is important to note that the underlying architecture is not restricted to accumulation and can be applied to any reduction operation circuit, such as multiply-reduce and min/max of a vector.

3.1 Partial Result Scheduling

To understand how partial results are scheduled to use the floating-point addition circuitry, it is important to note that, for an 8-or-9-stage pipelined adder, at most three unique streams, regardless of stream length, will ever be in buffer storage at a
given time. Consider the following scenarios for a
given stream length N:

- N < 8 - majority of their lifespan is spent in the
  floating-point core, requiring 2 unique streams to
  be tracked in storage at once (Figure 5a).
- 8 <= N <= 24 - there is more stress on the storage
  and scheduling units, resulting in 3 unique
  streams’ partial results residing in buffer storage
  (Figure 5b).
- N >= 24 - Partial reductions of stream k have
  been performed by the time stream k+1 enters the
  circuit, fully reduced before stream k+2 enters
  Figure 5c).

With this observation in mind, 3 copies of
tracking logic are created to manage operands in
buffer storage. Each copy contains some small state,
as well as some small FIFOs to record the location
of the stream’s current entries in the buffer storage.

3.2 Partial Result Storing

The nature of the scheduling scheme in this
architecture requires streams’ partial results to be
stored in-between reductions until further
processing can be done. Figure 6 demonstrates that
for 32-bit floating-point implementation, seven
buffers for the storage of intermediate data is

Figure 6: Buffers require for partial result storage

sufficient based on exhaustive simulations.

Figure 6 also shows that the storage for the
second operands actually requires 1 less buffer,
since once a given stream’s has data in both buffers,
it can be reduced further.

With regards to the 64-bit implementation, 2
additional buffers had to be added due to an extra
stage (9 vs. 8) in the 64-bit adder. However, because
the buffers were configured as Xilinx 16x1
distributed RAM, the extra storage is free. That is,
distributed RAM treats a 4-input LUT as a 1-bit
wide, 16-bit deep addressable memory. The only
extra cost is in scheduling these additional entries,
which is relatively inexpensive.

3.3 Multi-Stream Reductions

As Figure 8 shows, our particular implementation
requires tracking, at most, 6 unique streams at a
time during their reduction lifespan, requiring 3 bits of stream identification, or tag, to be piped through the circuit with the data.

A stream is fully reduced if the reductions sent to the adder from partial storage equal \((n / 2) - 1\). That is, a stream of length 4, would be fully reduced after three reductions: two fed from the input buffers and one fed from storage. Similarly, a stream of length 6 would be fully reduced after a total of 5 reductions; but only being fed 2 from storage.

4 Implementation

4.1 Implementation Details

<table>
<thead>
<tr>
<th>Pipeline Stages</th>
<th>FP64/128</th>
<th>FP64/256</th>
<th>FPACCUM092</th>
<th>FPACCUM154</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT Count</td>
<td>900</td>
<td>1282</td>
<td>1097</td>
<td>2088</td>
</tr>
<tr>
<td>Register Usage</td>
<td>3%</td>
<td>84%</td>
<td>79%</td>
<td>153%</td>
</tr>
<tr>
<td>Clock Speed MHz</td>
<td>152</td>
<td>1355</td>
<td>258</td>
<td>151</td>
</tr>
</tbody>
</table>

Both 32- and 64-bit floating point accumulation circuits were designed, tested, and implemented using Aldec Active-HDL, Synplify Pro 8.1, and Xilinx ISE 7.1i targeting our Virtex 2 8000-4 device. The underlying floating-point reduction cores are based on the IEEE 754 standard, and are presented in Table 1. It can be seen that the total accumulator design is a little over twice the size of their respective adder, which is more than tolerable in an FPGA platform.

4.2 Implementation Application

We have verified results using our hardware acceleration platform, the Celerity™ board [5]. To show the power of our accumulator circuit, we constructed a single-precision dense-Matrix-Matrix multiplier (referred to from here on as A*B=result), implemented for the previously described card.

The ‘A’ matrix was stored in DRAM, due to its large size and bandwidth, and was then streamed to the arithmetic units. When streaming data from the DRAM, the total bandwidth is 16, 32-bit words per clock cycle. Thus, we have chosen to have the system work on 4x4 blocks from the ‘A’ matrix simultaneously, and chose the number of columns from the ‘B’ matrix based on the amount of hardware available. Each cycle, we move horizontally through the ‘A’ matrix until the end of the row is reached. For simplicity, our initial implementation only supports square matrices.

The values from the ‘B’ matrix and the result matrix are all stored in DRAM along with the ‘A’ matrix. As our board supports 16 GB of DRAM, the maximum matrix size is 1.3 billion entries, or approximately 36,000x36,000 matrices. The BlockRAM (BRAM) in the chip (Virtex-II 8000) is used primarily to cache the working columns from the ‘B’ and result matrices. In our initial version, there was only sufficient BRAM to support the full caching of 6 columns of 7168 entries each, which imposes an artificial limitation. In later revisions, this limitation will be removed by allowing re-filling of the cache mid-column.

Each cycle, the 4x4 block from DRAM is multiplied by the appropriate entries from 6 different columns of the ‘B’ matrix, resulting in 96 multiplications. Each 1x4 strip is then reduced to one value by going through 3 adders. The first adder level reduces the 1x4 to 1x2, and the second level reduces this to a single value that is then sent to the accumulator for final reduction. The result is that 192 floating-point units are used in parallel.

The final performance of the system is quite impressive. As the multiplier, adder, and accumulator are perfectly pipelined, there are no stalls or delays and we accordingly get nearly-optimal performance. The performance can easily be shown to be: \(192 \times \text{clock speed} \times \text{efficiency factor}\). Our target clock speed is 150 MHz, which is reasonable given that we have shown the accumulator operating at 190 MHz in physical hardware. The efficiency factor is primarily dominated by DRAM refresh cycles and column turnaround. Simulations have shown this to be approximately 95% for a wide variety of problems. Combining these, we show an easily realizable throughput of 27.4 GFLOPs. Synthesis results show that this is attainable using approximately 83% of the total LUTs in the Virtex-II 8000 part.

5 Conclusion

The development of efficient, floating-point reduction operations, such as matrix-vector-multiply and vector dot-products, is becoming increasingly important as more application fields are turning to custom hardware implementations for algorithmic speedups. To this end, we have developed a
floating-point reduction unit that overcomes many of the shortcomings of previously published designs. In this paper, we presented our accumulation circuit architecture and determined its performance as part of a large matrix-matrix-multiply system capable of achieving a throughput of 27.4 GFLOPS.

References:


