Exploiting Asymmetry in Booth-Encoded Multipliers for Reduced Energy Multiplication

Mike O'Connor [§†] and Earl E. Swartzlander, Jr. [†]

[§] NVIDIA
11001 Lakeline Blvd., Building 2, Suite 100
Austin, TX 78717 USA

[†] Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712 USA

Email: moconnor@nvidia.com, eswartzla@ieee.org

***Abstract* – Booth Encoding is a common technique utilized in the design of high-speed multipliers. These multipliers typically encode just one operand of the multiplier, and this asymmetry results in different power characteristics as each input transitions to the next value in a pipelined design. Relative to the non-encoded input, changes on the Booth-encoded input induce more signal transitions requiring ~73% more multiplier array energy. This paper proposes low-overhead approaches to take advantage of this asymmetric behavior to reduce the energy of multiplication operations in pipelined SIMD architectures like GPUs. Compiler-based approaches that apply constant or uniform inputs to the Booth-encoded input of the multiplier can save 4.8% of multiplier energy on average. An additional 1.5% savings can be achieved with dynamic detection and steering of uniform inputs.**

I. Introduction

The energy required to perform multiplication operations is a significant source of power consumption in many important applications, ranging from deeply embedded DSPs to GPU-based accelerators used in the largest supercomputers. Booth Encoding [4] is a commonly employed technique in multiplier implementations because is reduces the number of partial products that must be summed using the carry-save adder tree. Radix-4 Modified Booth Encoding is often used in commercial multiplier designs [13], and it reduces the number of partial products by approximately a factor of two. The goals of these optimizations are primarily to minimize the delay and to reduce the area of the multiplier. One consequence of this approach, however, is that changes to the Booth-encoded input of the multiplier can induce significantly more switching activity than changes on the other, non-encoded, input. Earlier research [8] also observed this energy asymmetry in Booth-encoded multipliers.

Prior research has proposed some approaches to constructing reduced power multipliers based on reducing the switching activity resulting from Booth-encoded operands [6, 15]. Selecting the operand for encoding that results in the greatest number of encoded "zero" terms, described by [6], is well suited to integer and some fixed-point applications, but not widely applicable to floating point values which don't typically exhibit long strings of 0's or 1's. The approach described by [15] introduces latches to reduce spurious transitions generated by switching in the encoding path, but does not address the intrinsic additional switching that changes in the Booth-encoded input can induce.

This work explores techniques to minimize multiplier energy by optimizing the assignment of multiplier input parameters to the Booth-encoded operand. In particular, this work focuses on low-overhead techniques that can be exploited by pipelined, SIMD architectures like those found in many GPUs.

II. Multiplier Design

The multiplier design considered in this paper is commonly used in single-precision floating-point fused multiply-add units. In this work, only the 24×24-bit multiplication of the significand is considered. The additional exponent arithmetic, normalization, addition, and rounding logic found in these units is not considered. The approach described can be extended to larger (e.g. double-precision floating point or 32-bit integers) or smaller (e.g. 16-bit floating-point formats) multipliers in a straightforward manner.

The multiply operation takes two 24-bit values, A and B, and computes a 48-bit product. Generally, the multiplier logic consists of two primary components: Partial Product generation and a Carry-Save Adder.

A. Partial Product Generation

A Radix-4 Modified Booth Encoding scheme is assumed for partial product generation. The primary benefit of this approach is reducing the number of partial products that must be summed using the carry-save adder by a factor of two. The B input operand is encoded to determine the corresponding term by which the A input is multiplied. The encoder considers three bits at a time (b_{i+1}, b_i, and b_{i-1}) of the B operand, and generates three signals – *Neg*, *NotZero*, and *Shift* – according to Table I. Only the values where i is even are considered, and bit positions that fall "outside" the operand (e.g. b_{-1}, b_{24}, and b_{25}) are treated as having the value 0.

In the 24×24-bit multiplier, 13 partial product terms result from this radix-4 encoding. Each bit, i, of each term, j, of the partial product is then generated as:

$$A_masked_{i,j} = (a_i \text{ XOR } Neg_j) \text{ AND } NotZero_j$$
$$Partial_product_{i,j} = Shift_j \ ? \ A_masked_{i-1,j} : A_masked_{i,j}$$

Copyright 2015 IEEE. Published in the *Proceedings of the 49th Asilomar Conference on Signals, Systems, and Computer,s* Pacific Grove, CA, USA, Nov. 8-11,2015
Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

TABLE I

RADIX-4 MODIFIED BOOTH ENCODING

Pattern	Result	Neg	NotZero	Shift
000	0	0	0	0
001	+1	0	1	0
010	+1	0	1	0
011	+2	0	1	1
100	-2	1	1	1
101	-1	1	1	0
110	-1	1	1	0
111	0	0	0	0

In order to handle the negative partial product rows generated by the Booth encoder, the Sign-Generate Sign-Extension scheme described by [2] is used. The "single zero" Booth encoding scheme described in [2] is also used to reduce switching that would have been created by a "-0" term for the 111 pattern.

B. *Carry-Save Adder Generation*

The Carry-Save adder in the multiplier is generated with a Reduced Area Multiplier tree [3] using 3:2 (and 2:1) compressors. The inputs at each layer of the tree are selected in order to minimize spurious transitions. The design strives to match the delays of various inputs to each stage of the reduction. Sakuta, *et al*. [14] actively inserted delay elements to balance delays of the inputs to each stage. Rather than add gates with their associated area and power, a heuristic is employed that attempts select 2:1 compressor inputs that are at the same delay to that point, or 3:2 compressor inputs which two inputs are at the same level, and a third (corresponding to the "carry-in" term) is somewhat delayed. If no inputs meet these criteria, candidate inputs that have the least "spread" in estimated delay are selected. If multiple input sets meet the criteria, the lowest delay input set to that point is selected.

Alternative heuristics to determine the configuration of the CSA tree were evaluated including a transition-minimizing scheme suggested by Oskuii, *et al*. [12]. They select inputs at the upper levels of the CSA tree that are predicted to have the lowest likelihood of seeing a transition. Their approach searches the large number of permutations of possible mappings to find an optimal solution. This time-consuming approach is not well-suited for larger multipliers studied here. Instead, a greedy heuristic was employed that simply seeks to select the inputs with the lowest predicted probability of transition at each stage of the reduction is used. This scheme was determined to be most beneficial when it was used to "break ties" in the delay-balancing scheme described above.

These heuristics are applied at each level of the carry-save reduction before proceeding to the next. As a result, it is possible that some opportunities are missed. For instance, in the scheme used for this work, a subsequent level of reduction might expose a better opportunity for a balanced delay choice that was not possible at the earlier level, and that result might have been mapped to the input of another

TABLE II

24×24 MULTIPLIER OVERVIEW

Multiplier	Avg. Power (pJ)	Area (µm^2)
Radix-4 Modified Booth Encoding	18.17	3650.29

compressor which would have been better left to pair-up in the next reduction stage. The final carry-propagate addition is performed with a Kogge-Stone adder [9].

III. POWER ASYMMETRY OF BOOTH-ENCODED OPERANDS

The multiplier design is implemented using an open-source 45nm standard-cell library [11]. Using a simple wire-load model and the input-capacitance for each gate, the cells are appropriately sized such that high fan-out gates keep output transition times within acceptable limits. Power information based this standard-cell library is used to compute leakage and dynamic switching power for each gate. A gate-level discrete-event simulator simulates this multiplier for a sequence of pseudo-random numbers in a pipelined fashion, and captures signal transition and power information for each operation.

Using this infrastructure, the impact on the energy of the multiplier is evaluated as different input operands change at various rates. Figure 1 shows that when new pseudo-random inputs are applied to both inputs on each cycle, the Radix-4 multiplier requires an average 18.17 pJ of energy per multiply. If the rate at which the non-encoded, *A*, input operand changes is reduced such that a new pseudo-random value is provided only every 4 cycles (while the *B* input continues to change each cycle), the average energy of each multiply drops 13% to 15.82 pJ. If, on the other hand, the rate at which the Booth-encoded, *B*, input operand changes is reduced such that a new pseudo-random value is provided only every 4 cycles (while the *A* input continues to change each cycle), the average energy of each multiply sharply drops 41% to 10.81 pJ.

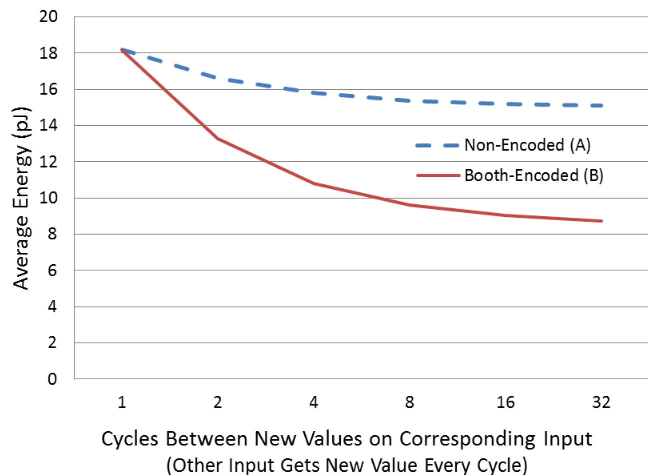

Fig. 1: Energy of 24×24 multiply as each operand varies at different rates.

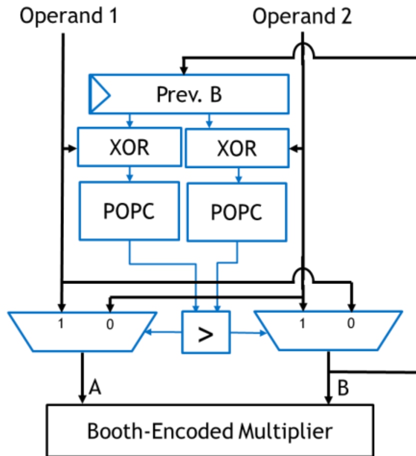

Fig. 2: Dynamic operand selection and steering.

Relative to the non-encoded input, changes on the Radix-4 Booth-encoded input induce ~73% more multiplier energy due to increased switching activity. A flip in a single bit in the Radix-4 Booth-encoded operand changes the encoded value of up to two partial product terms, potentially including shifts and negations of the non-encoded operand. Thus, a single bit in the Booth-encoded input operand can affect up to 54 bits in the partial-product inputs to the carry-save adder tree. Conversely, a flip of a single bit in the non-encoded input operand can flip at most 13 bits in the resulting partial-products. In addition, changes in the encoded input alter the values of the *Neg*, *NotZero*, and *Shift* outputs of the encoders. These signals fan-out to every bit in a partial-product term, requiring higher drive-strength gates, additional buffers, and longer wire. Variations in the arrival time of some of these signals also tend to introduce disproportionally more spurious transitions during the evaluation of the result. Thus, transitions in these signals are more expensive than the transitions in the partial product terms themselves.

IV. EXPLOITING MULTIPLIER POWER ASYMMETRY

Given the substantial asymmetry in the energy cost of changing the Booth-encoded operand relative to the non-encoded operand, higher-level architectural techniques exploiting this behavior are potentially attractive. Generally, schemes that minimize the switching activity on the Booth-encoded input will minimize the total multiplier power.

A. *Dynamic Operand Selection and Steering*

The most straight-forward "brute-force" approach to minimize the switching on the encoded input is simply comparing the Hamming distance of the new *A* and *B* input operands to the previous *B* input. If the Hamming distance of the new *A* input is less than the Hamming distance of the new *B* input, the *A* and *B* inputs are swapped. This approach is somewhat similar to a scheme described by Chen, *et al.* [5], in which the *A* or *B* input resulting in the fewest number of non-zero Booth-encoded terms is used as the Booth-encoded *B* input.

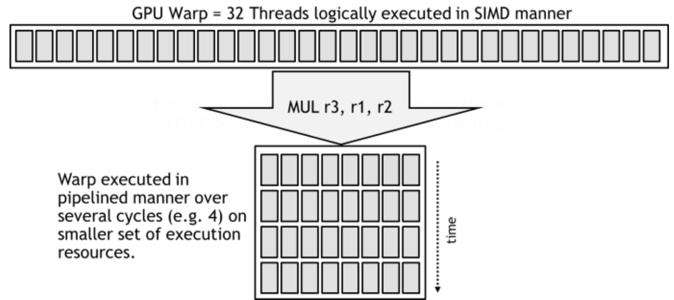

Fig. 3: Pipelined execution of SIMD operations in a GPU.

This approach requires additional latency and energy to compute the Hamming distance, reducing the potential savings within the multiplier. The design shown in Figure 2 was implemented and evaluated using the same infrastructure as the 24×24-bit multiplier. The computation and comparison of the Hamming distance, and the muxing of the operands requires 1.31pJ of additional energy per multiply. However, with pseudo-random inputs, this approach only saves an average of 0.44 pJ of multiplier power. This results for a net *increase* of 0.87 pJ per multiply. Clearly, techniques that require lower overheads to steer relatively static inputs to the Booth-encoded input of the multiplier are needed if this opportunity is to be exploited.

B. *Finding Static Opportunities*

The overheads of per-operation dynamic detection and selection of the best Booth-encoded operand make an approach that can statically determine the best operands attractive. Potential opportunities arise for this static approach in pipelined SIMD implementations, such as those found in GPUs, in which a multiply instruction is performed across the values in a SIMD (or vector) register.

As shown in Figure 3, GPU implementations may execute portions of a SIMD operation over several cycles in a pipelined manner. For instance, AMD's recent GPUs execute a 64-element SIMD instruction on a 16-wide execution pipeline over four cycles [1]. Proposed future GPU architectures use a *temporal SIMT* approach that execute the entire 32-thread GPU warp in a pipelined manner on a single execution unit over 32 cycles [16]. Thus, if a constant value or a uniform vector can be steered to the Booth-encoded multiplier input in these implementations, energy reductions similar to the 5.0 or 6.3 pJ/multiply savings shown in Figure 1 between the two bars at the columns labeled "4" and "32" can be realized.

Prior research has noted that within the SIMD execution engines of a GPU, a significant number of operands are scalar (uniform) across all the lanes of execution within a GPU [7]. Utilizing a cycle-accurate simulation framework designed for architectural exploration and performance analysis of current and future NVIDIA GPUs, statistics were gathered from 168 GPU application traces in which more than 5% of the dynamic instruction count consists of single-precision multiplications (or multiply-accumulates). These applications range across the various domains in which GPUs are applied. They include supercomputing, mobile,

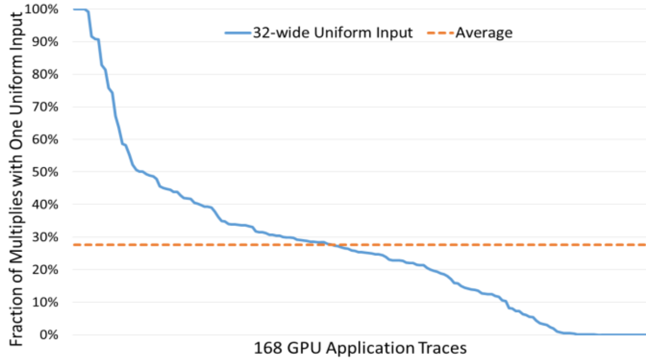

Fig. 4: Prevalence of single-precision multiplies with one uniform operand. (Overall average = 27.6%)

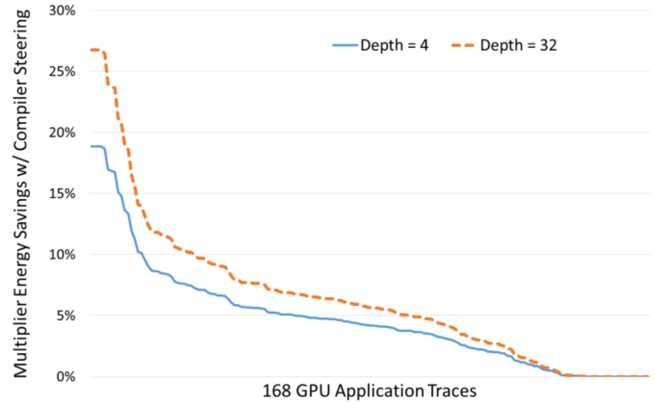

Fig. 5: Multiplier energy savings with compiler-driven steering. (Overall average = 4.8% w/ pipeline depth 4, 6.5% w/ pipeline depth 32)

workstation, PC gaming, and embedded image-processing workloads. Figure 4 shows, for each workload, the frequency of single-precision multiplies in which one operand is uniform across all 32-elements of a warp. One common reason for this situation is simply the case in which, for example, all elements in a vector are being multiplied by a constant value (e.g. π). The workloads that see ~100% of the multiplications as having one uniform input are either convolutions with a constant set of filter weights, or a kernel operating on a very sparse matrix in which the great majority of data values are all zero. Overall, an average of 27.6% of all single-precision multiplications have one uniform operand. If each of these can be steered to the Booth-encoded input of the multiplier, up to 12% of multiplication energy could be potentially saved.

C. *Compiler Selection of the Booth-encoded Input*

Given the opportunity, a low-overhead approach to identify and steer these uniform inputs to the encoded multiplier input is needed. Ideally, the compiler can determine which inputs are uniform or likely to be uniform. It can then specify these inputs as the Booth-encoded operand in the multiply instructions. This approach requires no additional hardware and no incremental energy cost to make the operand selection decision.

The first requirement to enable this compiler-driven operand steering technique is that the implementation must expose which multiplier source operand in the instruction-set architecture is mapped to the Booth-encoded input. Ideally, this would remain constant across different implementations of the ISA, allowing the same binary to be optimized for a variety of implementations. Instruction set architectures that allow constants to be specified as one operand of a multiplication operation should, obviously, also direct the constant value to the Booth-encoded input of the multiplier.

Next, the compiler must identify the scalar/uniform operands across a warp. Prior research [10] describes a compiler technique for *scalarization*. The scalarization optimization tries to identify instructions that produce identical results across for every element in a warp, and replace these instructions with a single scalar operation rather than the redundant warp-wide SIMD instruction. This occurs when the compiler can ensure that all the input registers to a warp-wide instruction are uniform. The same analysis that identifies these uniform registers can be leveraged to determine which multiplication operands are uniform and best suited to be applied to the Booth-encoded input of the multiplier.

This compiler approach was implemented to statically select uniform inputs to be steered to the Booth-encoded multiplier input. Figure 5 shows the potential energy savings across the 168 GPU workloads. The baseline to which it is compared is one in which there is a 50% chance that a given uniform input register happens to be applied to the Booth-encoded input. With an architecture that executes a warp pipelined over four cycles, the average energy savings is 4.8%. With a temporal SIMT implementation that pipelines a warp over 32 cycles, the energy savings is 6.5% on average.

D. *Dynamically Detecting Uniform Inputs*

Additional benefit can be gained from dynamic detection of uniform data. After steering data known at compile-time to be constant to the Booth-encoded input, additional opportunity remains. For example, data loaded from a sparse array may consist entirely of zeros, but the compiler will be unable to prove the result of the load operation is always uniform.

If logic exists to dynamically detect and flag uniform warp-wide registers (e.g. when data is returned from a load operation), then simple operand-muxing hardware can be added in front of the multiplier to ensure that uniform input registers are steered to the Booth-encoded input as shown in Figure 6. Using this approach provides additional opportunity to save multiplier energy. Figure 7 shows the additional savings beyond the compiler-only selection technique. Multiplier energy can be reduced another 1.5% on average in an implementation with warp execution pipelined over four cycles. An additional 2.1% savings can be achieved in a temporal SIMT implementation with a warp pipelined over 32 cycles.

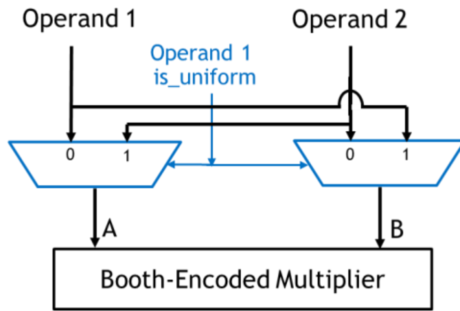

Fig. 6: Dynamic uniform input steering.

The additional logic that detects and marks uniform warp-wide registers requires additional area and energy, and the relatively minor additional savings in multiplier energy may not justify these overheads. The dynamic detection logic is also useful in scalarization [7], however. If this logic is present for other reasons, it can be easily leveraged to provide incremental multiplier energy benefits.

V. Conclusions

In this paper, the energy asymmetry of Booth-encoded multipliers is characterized in pipelined scenarios, showing that holding the Booth-encoded input constant rather than the other input can save up to 42% of multiplier energy. Exploiting this property in the context of pipelined SIMD processors like GPUs, a zero-overhead compiler-based approach for identifying uniform inputs and steering them to the Booth-encoded multiplier input is described. This approach saves up to 6.5% of single-precision multiplier energy on average in a temporal SIMT architecture which executes all 32-elements in a warp on a single, pipelined execution unit. An additional 2.1% reduction, for a total of 8.3% savings in this implementation, can be achieved if the architecture supports dynamic detection of uniform warp-wide registers.

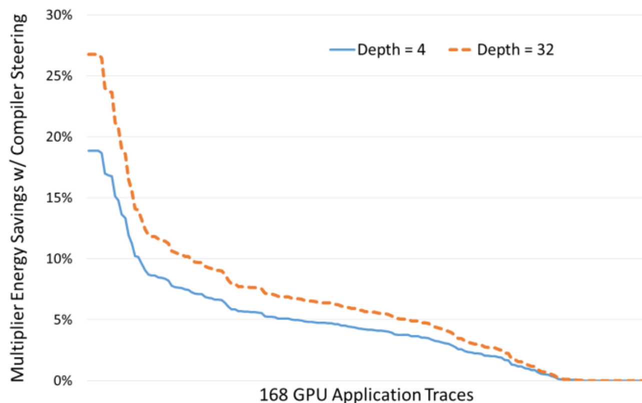

Fig. 7: Additional multiplier energy savings with dynamic detection and steering of scalar/uniform inputs.
(Overall average = 1.5% w/ pipeline depth 4, 2.1% w/ pipeline depth 32)

Additional benefits can also be gained by applying similar techniques to double-precision and integer multipliers. Also, while steering uniform data to the Booth-encoded input was the focus of this paper, potential other benefits can be derived from steering affine or other slowly varying data to the Booth-encoded input. Extending this approach to these areas and to traditional CPU architectures remains future work.

References

[1] Advanced Micro Devices, Inc, "AMD Graphics Core Next (GCN) Architecture White Paper," https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf , June 2012.

[2] E. de Angel and E.E. Swartzlander, Jr., "Low power parallel multipliers," *Workshop on VLSI Signal Processing*, 1996.

[3] K.C. Bickerstaff, M.J. Schulte, and E.E. Swartzlander, Jr., "Parallel reduced area multipliers," *Journal of VLSI Signal Processing Systems*, April 1995.

[4] A.D. Booth, "A Signed Binary Multiplication Technique," *Q. J. Mech. Appl. Math.* 1951.

[5] A.P. Chandrakasan and R.W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proc. IEEE*, Apr. 1995.

[6] O. T.-C. Chen, S. Wang, and Y.-W. Wu, "Minimization of switching activities of partial products for designing low-power multipliers," *IEEE Transactions on VLSI Systems*, June 2003.

[7] S. Collange, D. Defour, and Y. Zhang, "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computation," *Euro-Par 2009 – Parallel Processing Workshops*, 2009.

[8] K. Han, B.L. Evans, E.E. Swartzlander Jr., "Low-power multipliers with data wordlength reduction," *Asilomar Conference on Signals, Systems and Computers*. 2005.

[9] P.M. Kogge and H.S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, Aug. 1973.

[10] Y. Lee, R. Krashinsky, V. Grover, S.W. Keckler, and K. Asanovic, "Convergence and Scalarization for Data-parallel Architecture," *International Symposium on Code Generation and Optimization (CGO)*, 2013.

[11] *NanGate FreePDK45 Generic Open Cell Library* - https://www.si2.org/openeda.si2.org/projects/nangatelib

[12] S. Oskuii, P.G. Kjeldsberg, and O. Gustafsson, "Transition-activity Aware Design of Reduction-stages for Parallel Multipliers," *Great Lakes Symposium on VLSI*, 2007.

[13] E. Quinnell, *Floating-Point Fused Multiply-Add Architectures*, PhD Thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2007.

[14] T. Sakuta, Wai Lee, and P.T. Balsara, "Delay balanced multipliers for low power/low voltage DSP core," *IEEE Symposium on Low Power Electronics*, 1995.

[15] S. Saravanan and M. Madheswaran, "Design of low power multiplier with reduced spurious transition activity technique for wireless sensor network," *Fourth International Conference on Wireless Communication and Sensor Networks (WCSN)*, 2008

[16] O. Villa, D.R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, Peng Wang, P. Micikevicius, A. Scudiero, S.W. Keckler, and W.J. Dally, "Scaling the Power Wall: A Path to Exascale," *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.