University of Texas at El Paso
ScholarWorks@UTEP

Departmental Technical Reports (CS) | Computer Science

12-2000

Itanium's New Basic Operation of Fused Multiply-Add: Theoretical Explanation and Theoretical Challenge

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Follow this and additional works at: https://scholarworks.utep.edu/cs_techrep

 Part of the Computer Engineering Commons

Comments:
UTEP-CS-00-42.
Published in *ACM SIGACT News*, 2001, Vol. 32, No. 1 (118), pp. 115-117.

Recommended Citation

Kreinovich, Vladik, "Itanium's New Basic Operation of Fused Multiply-Add: Theoretical Explanation and Theoretical Challenge" (2000). *Departmental Technical Reports (CS)*. 503.
https://scholarworks.utep.edu/cs_techrep/503

This Article is brought to you for free and open access by the Computer Science at ScholarWorks@UTEP. It has been accepted for inclusion in Departmental Technical Reports (CS) by an authorized administrator of ScholarWorks@UTEP. For more information, please contact lweber@utep.edu.

Itanium's New Basic Operation of Fused Multiply-Add: Theoretical Explanation and Theoretical Challenge

Vladik Kreinovich*
Department of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA, vladik@cs.utep.edu

Abstract

A new Intel's 64-bit chip *Itanium* has a new instruction set which includes a fused multiply-add instruction $x_1 + x_2 \cdot x_3$ (see, e.g., [7]). In this short article, we explain the empirical reasons behind the choice of this instruction, give possible theoretical explanation for this choice, and mention a related theoretical challenge.

Empirical reasons behind the new basic operation. A natural reason for introducing a new basic operation is to speed up commonly occurring time-consuming computations. The selection of such operations was done a decade or so ago, when decisions were made which operations to implement in a (speedy) math co-processor. The main operation selected for this implementation is a *dot product* which transforms two arrays $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ into their dot (*scalar*) product $c = a_1 \cdot b_1 + \ldots + a_n \cdot b_n$. A natural sequential computation of the dot product consists of sequentially computing $s_1 := a_1 \cdot b_1$ and $s_k := s_{k-1} + a_k \cdot b_k$ for $k > 1$; the final result is s_n. On each stage of this computation, we need to compute the combination $s_{k_1} + a_k \cdot b_k$; it is therefore natural to hardware support this combination and thus, make its computation faster.

Another common time-consuming computational problem in which this combination is very helpful is the Horner's rule for computing the value of a polynomial. Here, $y = a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n$ is computed as $a_0 + x \cdot (a_1 + \ldots + x \cdot (a_{n-1} + a_n \cdot x) \ldots)$, i.e., by sequentially computing $s_{n-1} := a_{n-1} + a_n \cdot x$, and $s_i := a_i + x \cdot s_{i+1}$ for $i < n - 1$; the final result is s_0.

Theoretical explanation for the new operation: main idea. For many physical quantities x (e.g., for time, for temperature, etc.), its numerical value depends on the choice of the starting point. If we replace the original starting point with a new one which differs by a, then the original numerical value x is replaced by $x + a$.

In view of this possibility, it is reasonable to select data processing operations in such a way that when we make this shift $x \to x + a$ and want to redo the computations, we do not need to start "from scratch": the use of the old result should simplify the computations. Many standard elementary functions have this "shift-easiness" property:

- For example, if we have already computed $y = \exp(x)$, and we want to compute $z = \exp(x+a)$, then we do not need to apply a time-consuming exponentiation function again: it is sufficient to use a single multiplication $z := y \cdot \exp(a)$ (provided, of course, that the value $\exp(a)$ has been pre-computed).

*Copyright © Vladik Kreinovich, 2000

- Similarly, if we have computed $y_1 = \sin(x)$ and $y_2 = \cos(x)$, then we do not need to use time-consuming computation of trigonometric functions to find $z_1 = \sin(x+a)$ and $z_2 = \cos(x+a)$. It is sufficient to use two additions and four multiplications: $z_1 := \sin(a) \cdot y_2 + \cos(a) \cdot y_1$ and $z_2 := \cos(a) \cdot y_2 - \sin(a) \cdot y_1$.

- Similar formulas exist for $\tan(x)$, $\tanh(x)$, and for the non-linear "sigmoid activation function" $s(x) \stackrel{\text{def}}{=} 1/(1+\exp(-x))$ most commonly used in artificial neural networks (see, e.g., [1]). Moreover, for neural networks, the above "shift-easiness" requirement (plus a few reasonable additional properties) uniquely determines the activation function $s(x)$ and thus, explains its use [4, 5].

A similar "shift-easiness" property also holds for many functions of *discrete* variables: e.g., the combinatorial function $t(k) \stackrel{\text{def}}{=} \binom{n}{k}$ satisfies the property $t(k+1) = t(k) \cdot (n-k)/(k+1)$; so if we have already computed $y = t(k)$, we can compute $z = t(k+1)$ much faster than by re-computing it "from scratch". This "shift-easiness" property (exact term here is *hypergeometric*) is the main idea behind recent efficient algorithms for checking combinatorial-type inequalities [6] (see also [3]).

Theoretical explanation for the new operation: actual explanation. We can use the same "shift-easiness" argument not only in the analysis and selection of the *non-arithmetic* elementary functions, but also in the analysis and selection of the basic *arithmetic* operations.

Let us start with the simplest arithmetic operation: addition. Addition is already "shift-easy" in the above sense. Indeed, if we know $y = x_1 + x_2$, and we want to compute $z = (x_1 + a) + x_2$, then, instead of using two additions ($r := x_1 + a$ and $z := r + x_2$), we can use a single addition $z := y + a$. Similarly, subtraction is already "shift-easy" in the same sense.

For the next simplest arithmetic operation of multiplication, the situation is not that simple. If we know $y = x_1 \cdot x_2$, and we want to compute $z = (x_1 + a) \cdot x_2$, then we can still reformulate z in terms of y, as $z = y + a \cdot x_2$, but this reformulation does not seem to decrease the complexity of computing z:

- If we compute z "from scratch", we need two elementary operations – one multiplication and one addition: $r := x_1 + a$ and $z := r \cdot x_2$.

- On the other hand, to compute z from y, we still need two elementary operations – one multiplication and one addition: $r := a \cdot x_2$ and $z := y + r$.

This situation drastically changes is we implement the computation of $z := y + a \cdot x_2$ as a single hardware-supported operation: then:

- computing z "from scratch" requires two elementary operations, while

- computing z from y requires only a single operation.

Thus, the necessity to make computations "shift-easy" in the above sense naturally leads to the new operation $x_1 + x_2 \cdot x_3$.

It is worth mentioning that this new operation is already "shift-easy" in the new sense, so no further operations seem to be necessary:

- If we know $y = x_1 + x_2 \cdot x_3$, then $z = (x_1 + a) + x_2 \cdot x_3$ can be computed as $z := y + a$.

- If we know $y = x_1 + x_2 \cdot x_3$, then $z = x_1 + (x_2 + a) \cdot x_3$ can be computed by a single application of the new operation, as $z := y + a \cdot x_3$.

- Similarly, if we know $y = x_1 + x_2 \cdot x_3$, then $z = x_1 + x_2 \cdot (x_3 + a)$ can be computed by a single application of the new operation, as $z := y + a \cdot x_2$.

Related theoretical challenge. In algorithm design, the desire to have the fastest possible algorithms leads to a search for algorithms with the smallest *algebraic complexity* – usually understood as, e.g., a weighted combination of the total number of multiplications and of the total number of additions and subtractions (see, e.g., [2]).

The emergence of the new (fused) operation changes the objective, because now, the combination $x_1 + x_2 \cdot x_3$ requires only one elementary operation and not two as before. As a result, an algorithm which is known to be the best in the sense of traditional algebraic complexity may not be the best if we count this fused operation as one. It is therefore necessary:

- to revisit the corresponding algebraic computation problems,
- to check whether previously optimal algorithms are still optimal, and
- if the previously known algorithms are not optimal, to search for the algorithms which are optimal in the new sense – i.e., which contain the smallest possible number of "fused" operations.

Acknowledgments. This work was partially supported by NASA under cooperative agreement NCC5-209, and by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-00-1-0365.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic programming*, Athena Publ., Belmont, MA, 1996.

[2] Th. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, Cambridge, MA, and Mc-Graw Hill Co., N.Y., 1994.

[3] V. Kreinovich, "A review of [6]", *SIGACT News*, 2000, Vol. 31, No. 4 (December 2000), pp. 18–24.

[4] V. Kreinovich, O. Sirisaengtaksin, and H. T. Nguyen, "Sigmoid neurons are the safest against additive errors", *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, Atlanta, GA, May 28–31, 1995, Vol. 1, pp. 419–423.

[5] H. T. Nguyen and V. Kreinovich, *Applications of continuous mathematics to computer science*, Kluwer, Dordrecht, 1997.

[6] M. Petkovšek, H. S. Wilf, and D. Zeilberger, $A = B$, A. K. Peters, Ltd., Wellesley, MA, 1996.

[7] D. Scott, "Sixty-four bit architecture and the Itanium processor", *Abstracts of the 9th GAMM – IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics SCAN'2000, held jointly with the International Conference on Interval Methods in Science and Engineering Interval'2000*, Karlsruhe, Germany, September 19–22, 2000, p. 20.