ATLANTIS PRESS

Atlantis Studies in Uncertainty Modelling, volume 1

11th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2019)

Towards a More Efficient Representation of Functions in Quantum and Reversible Computing

Oscar Galindo[a] and Laxman Bokati[b] and Vladik Kreinovich[a,b]
[a]Department of Computer Science
[b]Computational Science Program
University of Texas at El Paso, El Paso, TX 79968, USA
ogalindomo@miners.utep.edu, lbokati@miners.utep.edu,
vladik@utep.edu

Abstract

Many practical problems necessitate faster and faster computations. Simple physical estimates show that eventually, to move beyond a limit caused by the speed of light restrictions on communication speed, we will need to use quantum – or, more generally, reversible – computing. Thus, we need to be able to transform the existing algorithms into a reversible form. Such transformation schemes exist. However, such schemes are not very efficient. Indeed, in general, when we write an algorithm, we composed it of several pre-existing modules. It would be nice to be able to similarly compose a reversible version of our algorithm from reversible versions of these moduli – but the existing transformation schemes cannot do it, they require that we, in effect, program everything "from scratch". It is therefore desirable to come up with alternative transformations, transformation that transform compositions into compositions and thus, transform a modular program in an efficient way – by utilizing transformed moduli. Such transformations are proposed in this paper.

Keywords: reversible computing, modular computing, quantum computing

1 Formulation of the Problem

Need for faster computing. While computers are very fast, in many practical problems, we need even faster computations. For example, we can, in principle, with high accuracy predict in which direction a deadly tornado will turn in the next 15 minutes, but this computation requires hours even on the most efficient high performance computers – too late for the resulting prediction to be of any use.

We can increase communication speed, but there is a limit. Faster computations means faster processing within each cell and faster communication between cells. At present, communication between cells is fast, but, potentially, it can be further increased. However, there is a limit to this increase: according to modern physics, all processes cannot move faster than the speed of light.

Because of this limit, it is desirable to have smaller processing units. For a chip of size ≈ 3 cm, the limitations on communication speed means that it takes at least 0.1 nanosecond (10^{-10} sec) for a signal to move from one side of the laptop to the other. This is close to the time ≈ 0.25 nanoseconds during this time a usual 4 GHz laptop performs an elementary operation. Thus, to further speed up computations, it is desirable to further decrease the size of the computer – and thus, to further decrease the size of its memory units and processing units.

Desirability of quantum computing. Already the size of a memory cell in a computer is compatible with the size of a molecule. If we decrease the computer cells even more, they will consist of a few dozen molecules. Thus, to describe the behavior of such cells, we will need to take into account the physical laws that describe such micro-objects – i.e., the laws of quantum physics.

Quantum computing means reversible computing. For macro-objects, we can observe irreversible processes: e.g., if we drop a china cup on a hard floor, it will break into pieces, and no physical process can combine these pieces back into the original whole cup. However, on the micro-level, all the equations are reversible. This is true for Newton's equations that describe the non-quantum motion of particles and bodies, this is true for Schroedinger's equation that takes into account quantum effects that describes this notion; see, e.g., [1, 3].

Thus, in quantum computing, all elementary operations must be reversible.

Reversible computing beyond quantum. Reversible computing is also needed for a different reason. Even at the present level of micro-miniaturization, theoretically, we could place more memory cells and processing cells into the same small volume if, instead of the current 2-D stacking of these cells into a planar chip, we could stack them in 3-D.

For example, if we have a terabyte of memory, i.e., 10^{12} cells in a 2-D arrangement, this means $10^6 \times 10^6$. If we could get a third dimension, we would be able to place $10^6 \times 10^6 \times 10^6 = 10^{18}$ cells in the same volume – million times more than now.

The reason why we cannot do it is that already modern computers emit a large amount of heat. Even with an intensive inside-computer cooling, a working laptop warms up so much that it is possible to be burned if you keep it in your lap. If instead of a single 2-D layer, we have several 2-D layers forming a 3-D structure, the amount of heat will increase so much that the computer will simply melt.

What causes this heat? One of the reasons may be design imperfections. Some part of this heat may be decreased by an appropriate engineering design. However, there is also a fundamental reason for this heat: Second Law of Thermodynamics, according to which, every time we have an irreversible process, heat is radiated, in the amount $T \cdot S$, where S is the entropy – i.e., in this case, the number of bits in information loss; see, e.g., [1, 3]. Basic logic operations (that underlie all computations) are irreversible. For example, when $a \,\&\, b$ is false, it could be that both a and b were false, it could be that one of them was false. Thus, the usual "and"-operation $(a, b) \to a \,\&\, b$ is not reversible.

So, to decrease the amount of heat, a natural idea is to use only reversible operations.

How operations are made reversible now? At present, in quantum (and reversible) computing, a bit-valued function $x_1, \ldots, x_n \to f(x_1, \ldots, x_n)$ of n bit-valued variables x_i is transformed into the following reversible operation:

$$T_f : (x_1, \ldots, x_n, x_0) \to (x_1, \ldots, x_n, x_0 \oplus f(x_1, \ldots, x_n)),$$

where x_0 is an auxiliary bit-valued variable, and $\oplus$ denotes "exclusive or", i.e., addition modulo 2; see, e.g., [2].

It is easy to see that the above operation is indeed reversible: indeed, if we apply it twice, we get the same input back:

$$T_f(T_f(x_1, \ldots, x_n, x_0)) =$$

$$T_f(x_1, \ldots, x_n, x_0 \oplus f(x_1, \ldots, x_n)) =$$

$$(x_1, \ldots, x_n, x_0 \oplus f(x_1, \ldots, x_n) \oplus f(x_1, \ldots, x_n)).$$

For addition modulo 2, $a \oplus a = 0$ for all a, so indeed

$$x_0 \oplus f(x_1, \ldots, x_n) \oplus f(x_1, \ldots, x_n) =$$

$$x_0 \oplus (f(x_1, \ldots, x_n) \oplus f(x_1, \ldots, x_n)) = x_0$$

and thus,

$$T_f(T_f(x_1, \ldots, x_n, x_0)) = (x_1, \ldots, x_n, x_0).$$

Limitations of the current reversible representation of functions. The main limitation of the above representation is related to the fact that we rarely write algorithms "from scratch", we usually use existing algorithms as building blocks.

For example, when we write a program for performing operations involving sines and cosines (e.g., a program for Fourier Transform), we do not write a new code for sines and cosines from scratch, we use standard algorithms for computing these trigonometric functions – algorithms contained in the corresponding compiler. Similarly, if in the process of solving a complex system of nonlinear equations, we need to solve an auxiliary system of linear equations, we usually do not write our own code for this task – we use existing efficient linear-system packages. In mathematical terms, we form the desired function as a composition of several existing functions.

From this viewpoint, if we want to make a complex algorithm – that consists of several moduli – reversible, it is desirable to be able to transform the reversible versions of these moduli into a reversible version of the whole algorithm. In other words, it is desirable to generate a reversible version of each function so that composition of functions would be transformed into composition. Unfortunately, this is not the case with with the existing scheme described above. Indeed, even in the simple case when we consider the composition $x_1 \to f(f(x_1))$ of the same function f of one variable, by applying the above transformation twice, we get – as we have shown – the same input x_1 back, and *not* the desired value $f(f(x_1))$.

Thus, if we use the currently used methodology to design a reversible version of a modularized algorithm, we cannot use the modular structure, we have, in effect, to rewrite the algorithm from scratch. This is not a very efficient idea.

Resulting challenge, and what we do in this paper. The above limitation shows that there is a need to come up with a different way of making a function

reversible, a way that would transform composition into composition. This way, we will have a more efficient way of making computations reversible.

This is exactly what we do in this paper.

2 Case of Fixed-Point Real Numbers: Analysis of the Problem and the Resulting Recommendation

Simplest case: description. Let us start with the simplest case of numerical algorithms, when we have a single real-valued input x and a single real-valued output y.

Of course, it is important to take into account that in a computer, we do not process actual real numbers (which form an infinite set), we process computer-representable real numbers – which form a finite set.

Let us denote the corresponding transformation by

$$x \to f(x).$$

In general, this transformation is not reversible. So, to make it reversible, we need to consider an auxiliary input variable u (and, if needed, more than one such auxiliary variable) – and, correspondingly, an auxiliary output variable v which depends, in general, on x and u: $v = v_f(x, u)$, for an appropriate function v_f. The resulting transformation $(x, u) \to (f(x), v_f(x, u))$ should be reversible.

In the actual computations, we will use a specific value u_0 of the auxiliary variable u.

How to make sure that composition is transformed into composition. We want to make sure that composition turns into composition. In other words, we want to make sure that if we have two functions f and g, then the result of transforming them one by one and the result of transforming their composition should be the same.

If we first apply the reversible analogue of the function f, then each original state (x, u_0) is transformed into $(y, v) \stackrel{\text{def}}{=} (f(x), v_f(x, u_0))$. To this new state, we would like to apply the reversible analogue of the second function g. We have agreed that the reversible analogue of applying a function should start with a state of the type (y, u_0). Thus, it is reasonable to require that $v_f(x, u_0) = u_0$ for all x (and for all functions f); then the state emerging after applying the reversible analogue of f will be $(y, u_0) = (f(x), u_0)$. If we now apply the reversible analogue of the function g to this new state, we get the state $(g(y), u_0) = (g(f(x)), u_0)$ – which is exactly what we would have got if, from the very beginning, we applied the reversible version of composition function $x \to g(f(x))$ to the original state (x, u_0).

One can easily see that without losing generality, we can assume, e.g., that $u_0 = 0$. Indeed, if we have a transformation $v_f(x, u)$ that uses a different value u_0, then, to get an equivalent transformation with $u_0 = 0$, we can do the following:

- first, we add u_0 to the original value u, thus replacing it with the new value $u_n \stackrel{\text{def}}{=} u + u_0$; this way, e.g., the original value $u = 0$ will become a new value $u_n = u_0$;
- apply the reversible analogue of the function f to the pair (x, u_n), thus getting the state

  $$(f(x), v_f(x, u_n) = (f(x), v_f(x, u + u_0));$$

  and
- finally, subtract u_0 from the resulting value v_n of the auxiliary variable, resulting in the state $(f(x), v)$ with

  $$v = v_n - u_0 = v_f(x, u + u_0) - u_0.$$

As a result, we get a new transformation function V_f for which

$$V_f(x, u) = v_f(x, u + u_0) - u_0.$$

In particular, for $u = 0$, we get

$$V_f(x, 0) = v_f(x, u_0) - u_0 = u_0 - u_0 = 0.$$

Thus, the corresponding analogue of the function f transform a pair $(x, 0)$ into the pair

$$(f(x), V_f(x, 0)) = (f(x), 0).$$

In view of this possibility, in the following text, we will assume that u_0 takes the simplest possible value

$$u_0 = 0.$$

What does "reversible" mean here? As we have mentioned earlier, in the computer, real numbers are represented with some accuracy. Because of this, there are finitely many possible computer representations of real numbers. In this section, we consider the case of fixed-point real numbers, when all the computer representations have the same accuracy ε; a more complex case of floating-point numbers, when different numbers are represented with different accuracy, will be analyzed in the next section.

Reversibility means that inputs and outputs are in 1-1 correspondence, and thus, for each 2-D region r, its image after the transformation $(x, u) \to (y, v)$ should contain exactly as many pairs as the original region r.

Each pair (x, u) of computer-representable real numbers takes the area of ε^2 in the (x, u)-plane. In each region of this plane, the number of possible computer-representable numbers is therefore proportional of the area of this region. Thus, reversibility implies that the transformation $(x, u) \to (f(x), v_f(x, u))$ should preserve the area.

Vice versa, let us assume that the transformation is area-preserving; let us show that this implies reversibility. Indeed, let $\delta > 0$ be the accuracy beyond which the different between inputs and/or outputs makes no physical difference. So, all the inputs within an area of size $\delta \times \delta$, are, in effect, practically equivalent. Since the area is preserved, the set of all corresponding outputs has the exact same area – and since these outputs correspond to practically equivalent inputs, they are also practically equivalent. Each of the two regions is formed by small $\varepsilon \times \varepsilon$ squares that correspond to machine accuracy (which is usually much higher than what we need in practice). Since the original area and its image have the same area, this means that they consist of the same number of such small squares. So, we can put these squares in 1-1 correspondence with each other – and thus, make the transformation reversible.

The same argument can be applied not only to our transformations $\mathbb{R}^2 \to \mathbb{R}^2$, but also to transformations $\mathbb{R}^n \to \mathbb{R}^n$ corresponding to any number of variables n. So, in this general case too, reversibility is equivalent to preserving the area.

From calculus, it is known that, in general, under a transformation

$$(x_1, \ldots, x_n) \to (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)),$$

the n-dimensional volume is multiplied by the determinant of the matrix with elements $\dfrac{\partial f_i}{\partial x_j}(x_1, \ldots, x_n)$. Thus, reversibility means that this determinant should be equal to 1.

Let us go back to our simple case. For the transformation $(x, u) \to (f(x), v_f(x, u))$, the matrix of the partial derivatives has the form

$$\begin{pmatrix} f'(x) & 0 \\ \dfrac{\partial v_f}{\partial x}(x, u) & \dfrac{\partial v_f}{\partial u}(x, u) \end{pmatrix},$$

where, as usual, $f'(x)$ denoted the derivative. Thus, equating the determinant of this matrix to 1 leads to the following formula

$$f'(x) \cdot \frac{\partial v_f}{\partial u}(x, u) = 1,$$

from which we conclude that

$$\frac{\partial v_f}{\partial u}(x, u) = \frac{1}{f'(x)}.$$

Thus,

$$v_f(x, U) = v_f(x, 0) + \int_0^U \frac{\partial v_f}{\partial u}(x, u)\, du =$$

$$v_f(x, 0) + \int_0^U \frac{1}{f'(x)}\, du =$$

$$v_f(x, 0) + \frac{U}{f'(x)}.$$

We know that $v_f(x, 0) = 0$, thus we have

$$v_f(x, u)(x, u) = \frac{u}{f'(x)},$$

and the transformation takes the form

$$(x, u) \to \left(f(x), \frac{u}{f'(x)}\right).$$

Examples.

- For $f(x) = \exp(x)$, we have $f'(x) = \exp(x)$ and thus, the reversible analogue is

$$(x, u) \to (\exp(x), u \cdot \exp(-x)).$$

- For $f(x) = \ln(x)$, we have $f'(x) = 1/x$ and thus, the reversible analogue is $(x, u) \to (x, u \cdot x)$.

Comment. The above formula cannot be directly applied when $f'(x) = 0$. In this case, since anyway, we consider all the numbers modulo the "machine zero" ε – the smallest positive number representable in a computer – we can replace $f'(x)$ with the machine zero. (And, by the way, infinities are OK in the IEEE standard 754 – they can be naturally processed.)

General case. Similarly, if we have a general transformation

$$(x_1, \ldots, x_n) \to f(x_1, \ldots, x_n) \stackrel{\text{def}}{=}$$

$$(f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)),$$

we want to add an auxiliary variable u and consider a transformation

$$(x_1, \ldots, x_n, u) \to$$

$(f_1(x_1,\dots,x_n),\dots,f_n(x_1,\dots,x_n),v_f(x_1,\dots,x_n,u))$. To make sure that composition is preserved, we should take $v_f(x_1,\dots,x_n,0)=0$. Thus, from the requirement that the volume is preserved, we conclude that

$$v_f(x_1,\dots,x_n,u)=\frac{u}{\det\left\|\dfrac{\partial f_i}{\partial x_j}(x_1,\dots,x_n)\right\|}.$$

Resulting recommendation. To make the transformation

$$(x_1,\dots,x_n)\to(f_1(x_1,\dots,x_n),\dots,f_n(x_1,\dots,x_n))$$

reversible, we should consider the the following mapping:

$$(x_1,\dots,x_n,u)\to$$

$$\left(f_1(x_1,\dots,x_n),\dots,f_n(x_1,\dots,x_n),\frac{u}{\det\left\|\dfrac{\partial f_i}{\partial x_j}\right\|}\right).$$

3 Case of Floating-Point Numbers

In the previous section, we considered only fixed-point real numbers, for which the approximation accuracy ε – the upper bound on the difference between the actual number and its computer representation – is the same for all possible values x_i.

In some computations, however, we need to use floating-point numbers, in which instead of directly representing each number as a binary fraction, we, crudely speaking, represent its logarithm: e.g., in the decimal case, 1 000 000 000 is represented as 10^9, where 9 is the decimal logarithm of the original number. In this case, we represent all these logarithms with the same accuracy ε. In this case, the volume should be preserved for the transformation of logarithms $\ln(x_i)$ into logarithms $\ln(f_j)$, for which

$$\frac{\partial\ln(f_i)}{\partial\ln(x_j)}=\frac{x_j}{f_i}\cdot\frac{\partial f_i}{\partial x_j}.$$

In this case, formulas similar to the 1-D case imply that the resulting reversible version has the form

$$(x_1,\dots,x_n,u)\to$$

$$\left(f_1(x_1,\dots,x_n),\dots,f_n(x_1,\dots,x_n),\frac{u}{\det\left\|\dfrac{x_j}{f_i}\cdot\dfrac{\partial f_i}{\partial x_j}\right\|}\right).$$

In some cases, the input is a fixed-point number while the output is a floating point number; this happens, e.g., for $f(x)=\exp(x)$ when the input x is sufficiently large. In this case, we need to consider the dependence of $\ln(f)$ of x.

4 Case of Functions of Two Variables

If we are interested in a single function f of two variables, then it makes sense not to add an extra input, but to add an extra output, i.e., to consider a mapping $(x_1,x_2)\to(f(x_1,x_2),g(x_1,x_2))$, for an appropriate function g.

The condition that the volume is preserved under this transformation means that

$$\frac{\partial f}{\partial x_1}\cdot\frac{\partial g}{\partial x_2}-\frac{\partial f}{\partial x_2}\cdot\frac{\partial g}{\partial x_1}=1.$$

For example, for the function $f(x_1,x_2)=x_1+x_2$, we get the condition

$$\frac{\partial g}{\partial x_2}-\frac{\partial g}{\partial x_1}=1.$$

This expression can be simplified if, instead of the original variables x_1 and x_2, we use new variables $u_1=x_1-x_2$ and $u_2=x_1+x_2$ for which $x_1=\dfrac{u_1+u_2}{2}$ and $x_2=\dfrac{u_2-u_1}{2}$. In terms of the new variables, the original function g takes the form

$$g(x_1,x_2)=G(u_1,u_2)\stackrel{\text{def}}{=}g\left(\frac{u_1+u_2}{2},\frac{u_2-u_1}{2}\right).$$

For this new function G, we have:

$$\frac{\partial G}{\partial u_1}=\frac{1}{2}\cdot\frac{\partial g}{\partial x_1}-\frac{1}{2}\cdot\frac{\partial g}{\partial x_2}=-\frac{1}{2}.$$

Thus,

$$G(u_1,u_2)=-\frac{1}{2}\cdot u_1+C(u_2)$$

for some function C, i.e., substituting the expressions for u_i in terms of x_1 and x_2,

$$g(x_1,x_2)=\frac{x_2-x_1}{2}+C(x_1+x_2).$$

So, to make addition reversible, we may want to have subtraction – the operation inverse to addition; this makes intuitive sense.

Similarly, for the function $f(x_1,x_2)=x_1\cdot x_2$, we get the condition

$$x_2\cdot\frac{\partial g}{\partial x_2}-x_1\cdot\frac{\partial g}{\partial x_1}=1.$$

This expression can be simplified if we realize that $x_i\cdot\dfrac{\partial f}{\partial x_i}=\dfrac{\partial f}{\partial X_i}$, where we denoted $X_i\stackrel{\text{def}}{=}\ln(x_i)$. In these terms, we have

$$\frac{\partial g}{\partial X_2}-\frac{\partial g}{\partial X_1}=1,$$

and thus, as in the sum example, we get

$$g(X_1, X_2) = \frac{X_2 - X_1}{2} + C(X_1 + X_2).$$

Thus, we get

$$g(x_1, x_2) = \frac{\ln(x_2) - \ln(x_1)}{2} + C(\ln(x_1) + \ln(x_2)),$$

i.e.,

$$f(x_1, x_2) = \frac{1}{2} \cdot \ln\left(\frac{x_2}{x_1}\right) + C(x_1 \cdot x_2).$$

So, to make multiplication reversible, we need to add a (function of) division – the operation inverse to multiplication. This also makes intuitive sense.

Acknowledgments

This work was partially supported by the US National Science Foundation via grant HRD-1242122 (Cyber-ShARE Center of Excellence).

The authors are thankful to the anonymous referees for valuable suggestions.

References

[1] R. Feynman, R. Leighton, and M. Sands, *The Feynman Lectures on Physics*, Addison Wesley, Boston, Massachusetts, 2005.

[2] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, 2000.

[3] K. S. Thorne and R. D. Blandford, *Modern Classical Physics: Optics, Fluids, Plasmas, Elasticity, Relativity, and Statistical Physics*, Princeton University Press, Princeton, New Jersey, 2017.