PARALLEL FIXED POINT DIGITAL DIFFERENTIAL ANALYZER

Ramón P. Mollá, Ricardo Quirós, Javier Lluch, Roberto Vivó.
Sección de Informática Gráfica.
Departamento de Sistemas Informáticos y Computación.
Universidad Politécnica de Valencia.
Camino de Vera, 14
46071 Valencia SPAIN
e-mail: rmolla@tierra.upv.es, rquiros@dsic.upv.es
Tel.+(34) 6 3877351
FAX: +(34) 6 3877359

ABSTRACT

Two main serial algorithms to scan convert straight lines have been proposed: Bresenham and Digital Differential Analyzer.The Bresenham algorithm has became a standard because of integer arithmetic. Many theoretical solutions have been proposed to parallelize Bresenham algorithm but its implementation is difficult. So most parallelizations take advantage of repeated patterns, massive parallel computers and so on. Sequential Digital Differential Analyzer shows better peformance than Bresenham if fixed point arithmetic is used. This algorithm can be pipelined and parallelized. It is easily hardware implemented and scalable. Hardware cost is linear with speed up. Utilization is nearly 100% and hardware waste is low.

KEY WORDS : Digital Differential Analyzer, Line drawing, Fixed Point Arithmetic, parallelization, graphic coprocessors.

INTRODUCTION.

Scan conversion of straight line segments in a frame buffer is an important problem to solve in any computer graphics system. Although Bresenham's algorithm [1] can generate line segments at rates of more than one million pixels per second on many graphics workstations, many applications require even higher speeds. Since this algorithm is quite optimized, parallelization becomes the best solution for perfomance increase. Several methods to speed up Bresenham's algorithm have been tried using parallelization techniques [2][6], or by trying to take advantage of the repeated patterns that the algorithm generates [3][4] or by mixing both methods [5].

In the solutions given above, speed up can be constrained to three main problems:

·the dependencies graph avoids short steps or
·the average number of active steps is reduced.
·when the ratio (amount of operators) / (speed up) is not linear, the hardware costs can be prohibitive if speed up is high due to the amount of operators needed.

When we try to obtain a parallel version of an incremental algorithm such as Digital Differential Analyzer (D.D.A.), we can see that the number of operators has a linear dependence with the speed up but each operator is complex since floating point arithmetic is used. This is the main reason why this algorithm has been inadequate for hardware implementation. So the key matter is to diminish the hardware and timing cost of each operator. This can be done if calculations use fixed point (F.P.D.D.A.) instead of floating point arithmetic [7].

ALGORITHM DESCRIPTION

Let's suppose that we have sampled a point Pi (xi, yi) and the next *n* consecutive line points are to be drawn. Assuming without loss of generality that line slope *m* belongs to [-1, 1], the next *n* line points can be calculated by

$P_{i+1} = (X_i+1,\ Y_i+m);$
$P_{i+2} = (X_i+2,\ Y_i+2*m);$
....
$P_{i+n} = (X_i+n,\ Y_i+n*m);$

As in Bresenham's algorithm, an Initialization Phase (I.P.) is neccesary to detect the scan direction (X swept when *the line slope* $m \in [-1,1]$ or Y swept when $m \in ([-1,-\infty] \cup [1,\infty])$). After obtaining the scan direction, this phase uses a divider to calculate the slope *m* and some registers to save initial points from where to start line drawing in the next phase. In this parallel version, an extra array of adders and wired shifters is needed to calculate $j*m$ for every F.P.D.D.A. operator.

In the Loop Phase *n* F.P.D.D.A. operators are needed to calculate the next *n* line points in paralel. So the operator *j* would have to

Add *j* to X_i.
Multiply *j* by *m*.
Add $j*m$ to Y_i.

The last point calculated P_{i+n}, is used as P_i in the next loop step. When the last point sent to video memory is detected, the Loop Phase is finished and another line drawing can be achieved.

A block diagram of this algorithm can be analyzed in Fig. 1. A parallel pseudocode implementation of the described method is given below:

```
Procedure PFPDDA (int X0, Y0, Xf, Yf, color)
BEGIN
    CONST ONEFP 11 //A 1024 frame buffer,  need 11 decimal bits to avoid excessive errors.
    INTEGER Ax, Xinic, Xfinal;
    LONG INTEGER Ay,  Yinic, m, slope[n], x[n], y[n];

    Ax = Xf - X0;
    Ay = (Yf - Y0) << ONEFP; //This shift translate from Integer format to Fixed Point Format

    IF X0 > Xf
    THEN     Xinic = Xf; Xfinal = X0;
              Yinic  = (Yf + 0.5) << ONEFP;
    ELSE     Xinic = X0; Xfinal = Xf;
              Yinic  = (Y0 + 0.5) << ONEFP;
    ENDIF

    //Slope Calculation.
    m = Ay / Ax;      //Integer division.

    //Multiple Slope Calculation.
    ParFor i=0 TO n-1 //Paralell FOR of n operators.
    BEGIN slope [i] = i*m; ENDParFor

    plot (Xinic, Yinic, color);

    //Paralell Loop Phase
    WHILE Xinic < Xfinal
    BEGIN
        PARALELL //FPDDA operators
            BLOCK1
            ParFor i=0 TO n-1
            BEGIN
            x[i] = Xinic + i;
            y[i] = Yinic + slope[i];
            ENDParFor
            Yinic = y[n-1];
            Xinic = x[n-1];
            ENDBLOCK1

            BLOCK2 //Queue Manager. Serializer.
            FOR i=0 TO n-1
             IF x[i] <> Xfinal
             THEN plot (x[i], y[i], color);
             ENDIF
            ENDBLOCK2
        ENDPARALELL
    END
END
```

IMPLEMENTATION

Initialization Phase (I.P.)

Let's do $k = \log_2 (\max (N,M))$, where N and M are the length and width of a frame buffer sized in pixels.

When we want to see a real scene on the screen, a projection of all objects is made

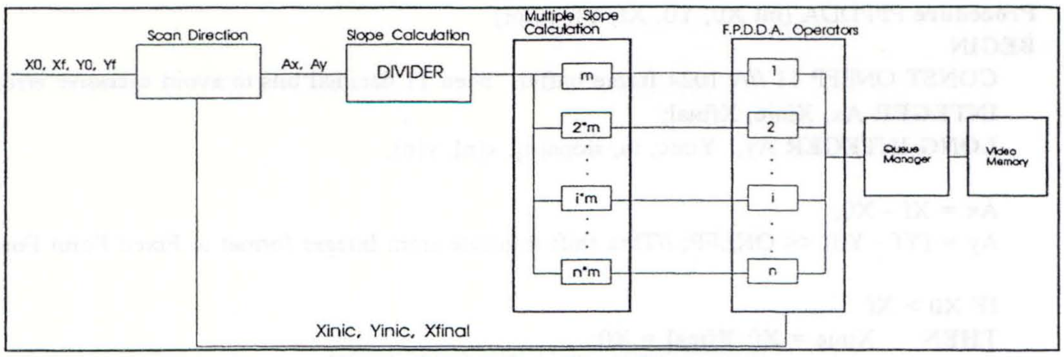

Fig. 1. P.F.P.D.D.A. Block Diagram

towards the screen plane. Intersections with the screen plane are floating point numbers. To represent a line that joints two intersection points, a round operation must be done to translate line ends to the screen integer coordinates since a raster is a discrete device. This round operation introduces an error of 0.5 pixels in the worst case. But if *k*+1 decimal bits are necessary to perform F.P.D.D.A. [7], this round operation can be shifted to the least significative bit, reducing the starting error 2^{k+1} times and producing a more accurately line representation. This phase, as shown in Fig. 1, has three well defined parts:

·Scan Direction.
·Slope Calculation.
·Multiple Slope Calculation.

Scan Direction.

Both Bresenham and D.D.A. algorithms have a similar I.P. In a first step, width (Δx) and height (Δy) are obtained. The difference between them points to the scan direction. Using this direction, initial points in the next phase are got easily.

From the beginning of this phase until this point, both Bresenham's and D.D.A. have the **same** time delays.

Since Bresenham's algorithm need East increment, North-East and error function F_{xy}, the hardware requirements are bigger than D.D.A. at a first glance because more latches and adders are needed. Nevertheless Bresenham's time delay is increased only in one addition since many extra operations can be overlapped.

All adders, buffers and buses used in this part are 2*k*+1 bits wide. Assuming that a cicle is one gate delay, the drawing direction can be calculated in less than 30 cycles aproximately asuming k=12 (2048x2048 frame buffer).

Slope Calculation.

Since D.D.A. needs the line slope to work, an integer division must be performed to obtain it. This is the main drawback of the D.D.A. algorithm because it increases I.P. considerably.

The integer division has a dividend of 2k+1 bits and the divisor k bits. The result is k+1 bits wide. Assuming no acceleration, *k*+1 adders of *k* bits are needed to perform this operation.

Multiple Slope Calculation

As it has been seen before, in every step during the Loop Phase, the *j* operator must multiply *j* by *m* in order to add it to P_i and obtain the *y* coordinate of P_{i+j}. Since both *j* and *m* are constant, an improved version could do this product and store it in a register. In this case a multiplication would be saved in each step during the Loop Phase

For this reason, in this parallel version, an extra array of adders and wired shifters are used to calculate $j*m$ for every D.D.A. operator.

As it can be seen in [7], these multiplications can be done using only wired shifters and adders if fixed point arithmetic is used in order to obtain an Accelerated Multiplication Circuit (A.M.C.).

If $l = \log_2(n)$, this multiplier can be accomplished using less than 2^{l-1} adders, that is to say, $n/2$ adders in the worst case. Temporal cost is l-2 additions if $l \geq 4$, l-1 for $3 \geq l \geq 2$, and 0 for l=1.

Loop Phase.

This part is called Loop Phase because a paralel loop of n F.P.D.D.A. operators works to obtain n consecutive line pixels. This phase is composed of two main concurrent blocks that work in a pipelined fashion:

·F.P.D.D.A. operators.
·Queue Manager.

F.P.D.D.A. Operators (F.O.)

Input data for this phase is the line slope array of registers. The i-th element of this array stores the value $i*m$ and is assigned to the i-th Y-operator. These values were calculated during the MSC phase by the multiplier. Other input data is the first line point to draw. Its coordinates are stored in two registers called X_{init} and Y_{init}. The X coordinate of the last point to draw is stored in a register called X_{final}. This register is used to detect the end of the Loop Phase Every F.P.D.D.A. operator is compound of two suboperators called X-operator and Y-operator. The former calculates the pixel X coordinates and the latter the Y ones.

The j-th X-operator uses an adder to increase X_{init} in j pixels. Once this new coordinate has been obtained, it´s inmediately compared to the X_{final} register. If they are equal, a 1 is stored in its status bit and vice versa. In any case, $X_{init} + j$ is stored in a register. So, the X-operator performs one addition of k bits, one k bits comparison and a register load.

The *j-th* Y-operator adds the j position of the slope register array to Y_{init} to obtain $Y_{init}+j*m$. This operation cost one addition of $2k$+1 bits.

If one of the status bits is set, then the FPDDA Operators phase is over and if there is available data, another line starts to be drawn while the Queue Manager finishes to extract last line final points to the video memory.

Queue Manager (Q.M.)

The first time the initial point coordinates are stored in X_{init} and Y_{init} registers, they are sent to video memory without checking them since the line is at least one pixel long.

When the control circuits detect that the new coordinates have been calculated, a register load signal is activated. A n state machine begins to extract these coordinates to video memory. The scan bit (calculated in the Initialisation Phase) helps to form the video memory address.

When the status bit of a X-register is set, it means this is the last point to draw. So this point will be sent to video memory and the queue manager cicle will be finished. The Queue Manager will remain idle until another line command be ordered and calculated.

The latter sends the pixels calculated by the FPDDA Operators, to the video memory. Meanwhile the former calculates the new pixels coordinates. When calculations are finished and pixels have been sent to memory, a load of intermediate registers is ordered and both blocks can go on.

VALIDATION

In order to see the differences between Bresenham's and FPDDA algorithms a benchmark of 10,000 lines were generated. 100 packets of 100 lines were obtained. Only length and slope line changed from one packet to other. Line length was incremented every time by 100 pixels. Line slope was incremented by 0.1

Pixel coordinates given by both Bresenham and FPDDA algorithms were compared to the real ones. For every 100 lines packet, average errors were saved. The number of exactly equal lines and the average number of different pixels between Bresenham's and FPDDA algorithm were also saved.

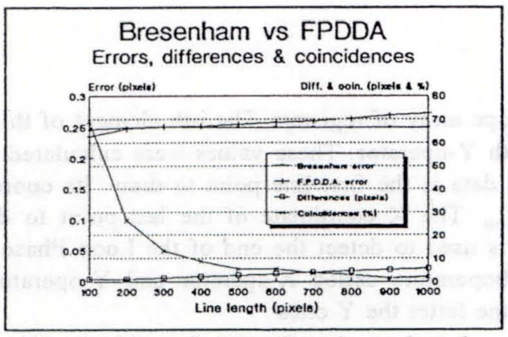

Fig. 2a. Normal slope. Results vs length.

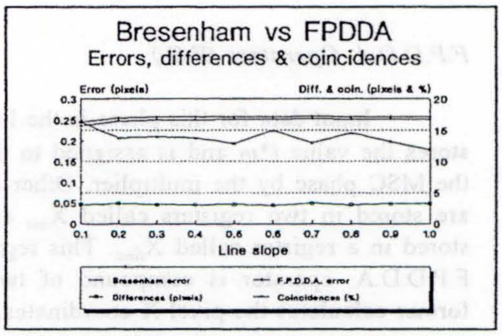

Fig. 2.b. Normal slope. Results vs slope.

The results can be seen in Fig. 2. While Bresenham's algorithm is much more accurate than FPDDA, the average differences are only 5% in the worst cases. While FPDDA errors are lower or equal than Bresenham's when line length is inferior to 350 pixels, this error starts to increase slowly. When line length is over 1000 pixels this error is nearly 7%. Errors were expressed in pixels. This can be also proved if we take a glance at the average number of different pixels per line. It increases when the line length does. The percentage of identical lines when line length is short is quite high (>70%) and it decreases until 2% when line length arrives

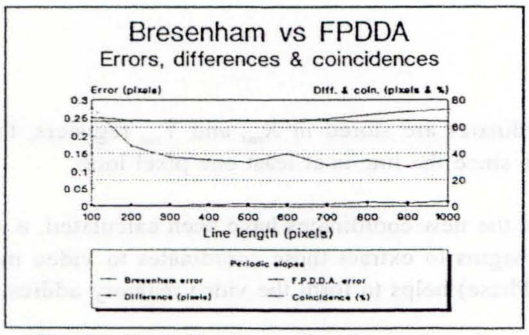

Fig. 2c. Periodic slope. Results vs length.

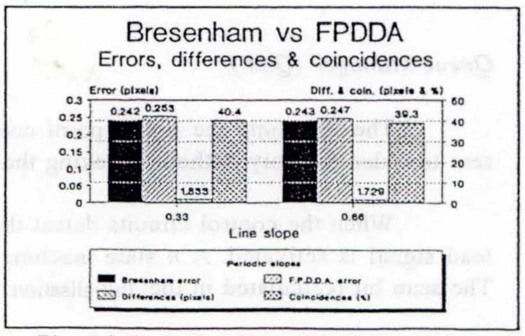

Fig. 2d. Periodic slope. Results vs slope.

to 1000 pixels.

If these parameters were compared to the line slope, we could see that they were more uniformly distributed. These results were better for periodic slopes (0.3 and 0.6).

SIMULATION RESULTS

Since the available circuit design tools didn't allow us to implement the whole P.F.P.D.D.A. in a single circuit,it was split into basic operators and was simulated taking into account real timing.

The simulations were done assuming no pipelining in the Initialisation Phase, althougth the Initialisation Phase could be pipelined with the FPDDA Operators and the Queue Manager Nevertheless, if pipelining would have been used, the Initialisation Phase average time delay would have been reduced and better results would have been obtained. So these graphics are the worst case results. They can be seen in Fig. 3.

The Fig. 3a. shows the FPDDA Operators and the Queue Manager phases utilization when no pipelining is assumed between them and the Initialisation Phase As it could be thought, the Initialisation Phase overhead is high when line lengths are short. As the line length increased, F.P.D.D.A. Operators were more time used and

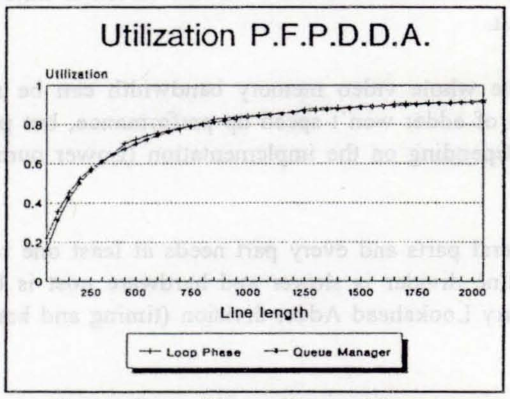

Fig 3a. Loop Phase, Queue Manager Utilization

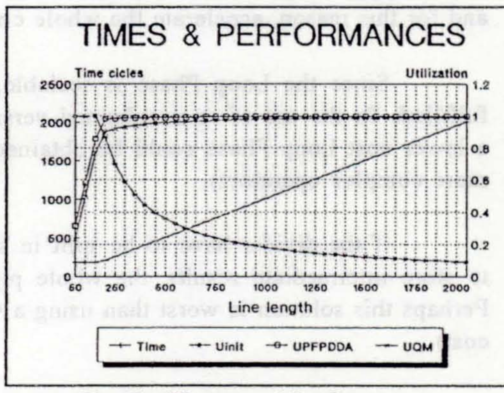

Fig 3b. Times and Performances.

this constant overhead reduced the proportion. So utilization and performance incresed with line length.

Fig. 3b. shows the same parameters as Fig. 3a. when Initialisation Phase and F.P.D.D.A. Operators and Queue Manager pipelining was assumed. Initialisation Phase utilization can also be seen.

Initialisation . Phase was constant and bigger than the F.P.D.D.A. Operators and Queue Manager phases when line length was relatively short. We assuméd an Initialisation Phase delay of 200 gates. We also assumed that the F.P.D.D.A. Operators and Queue Manager could send a line point to the video memory every gate delay. For these reasons when line length sized in pixels was lower than Initialisation Phase sized in gate delays, the timing cost was always equal to the Initialisation Phase time delay. But when line lengths were longer, the timing cost sized in gate delays was equal to the line length, since the Initialisation Phase was completely embedded in the F.P.D.D.A. Operators and Queue Manager phase.

CONCLUSIONS AND IMPROVEMENTS

Following a top-down methodology, we can see that pipeline is used to overlap both

Initialisation Phase and L.P. Pipeline is also used in the L.P. to overlap F.P.D.D.A. Operators and Queue Manager An array of operators is used to increase speed up working all together in the F.P.D.D.A. Operators block. Little more can be done to accelerate these steps.

On the other hand, Initialisation Phase can be accelerated using pipelining. The Initialisation Phase may be split into three steps:

·Initial Points and Scan Direction (IPSD).
·Slope Division (SD).
·Accelerated Multiplication Circuit (AMC).

IPSD time delay is no more than two or three adders and some glue logic. multiplier last no more than $\log_2(n)$-2 adders delay. Normally n is eight or sixteen, so multiplier spends no more than two adders delay, what is more or less the same as IPSD phase. Bottleneck is, of course, the integer division of $(2k+1)\mathrm{x}k$ bits. This phase uses no less than $k+1$ (usually 12 or 13) adders. But this phase can be pipelined also to reduce the division to steps of three adders time delay, avoiding the bottleneck and aproximating the initialization phase timing cost to Bresenham's one (three adders and a few glue logic).

Another way of speeding up the circuit is to use faster operators. As we have seen before, the adder is the basic unit to implement both Initialisation Phase and Loop Phase Apart from using pipeline or not, we can use Carry Lookahead Adders to reduce addition time delay, and for this reason, accelerate the whole circuit.

Since the Loop Phase is scalable, the whole video memory bandwidth can be always fulfilled. So the use of an accelerated version of adder won't speed up performance, but perhaps a lower cost Loop Phase could be obtained depending on the implementation (Lower number of more complex operators).

If the divider have to be split in several parts and every part needs at least one register to store intermediate results, the whole pipeline divider is slower and hardware cost is bigger. Perhaps this solution is worst than using a Carry Lookahead Adder division (timing and hardware cost).

Looking at the *sequential* implementation, some points are remarkable comparing with Bresenham's algorithm

·There are **no floating point** operations.
·There are only integer additions and comparisons in the main loop.
·Initialization is *relatively* short. There are only one integer division and a few integer additions and shifts.
·Easier mathematical tools are needed to understand and demonstrate the algorithm, so comprenhension and error correction are eased.
·Hardware operators are even simpler.
·An average of 0.25 to 0.5 additions and 1 comparison are saved in each loop step.

When the parallel version is analyzed, some points are remarkable:

·If the chip technology used allows a gate delay lower than a video memory write cicle, the PFPDDA can be designed to use the whole video memory bandwidth.
·The FPDDA algorithm produces an error slightly higher than Bresenham's algorithm but this difference is not significative at all.
·Since the PFPDDA circuit is scalable, speed-up can be increased as much as technology can support. In the theorical limit, asuming no technology limitations, a line could be drawn in a logarithm time with a linear hardware cost. These afirmations are relative to the line length.
·Operators utilization increases with line length.
·The use of the simplest operators (integer adders and comparators or registers) makes it suitable for hardware implementation. This simplicity saves design time and hardware.

·Pipelining increases productivity, chip utilization and speed up.
·This PFPDDA has a speed-up/(hardware cost) ratio constant.

REFERENCES

[1] Bresenham, J.E., "Algorithm for Computer Control of a Digital Plotter", IBM Systems Journal, Vol. 4, No. 1, 25-30. 1965

[2] Wright, W. E., "Paralellization of Bresenham´s Line and Circle Algorithms", IEEE CG&A, Vol. 10, No. 5, Pag. 60-67. 1990

[3] Earnshaw, R. A., "Line Tracking for Incremental Plotters", The Computer Journal, Vol. 23, No. 1, Pag. 46-52. 1980

[4] Castle, C. M. A., Pitteway M. L. V., "An Applications of Euclid´s Algorithm to Drawing Straight Lines", Fundamentals Algorithm for Computer Graphics, NATO ASI F17, Springer-Verlag, Pag. 134-139, 1985

[5] Angel, E., Morrison, D. "Speeding up Bresenham´s Algorithm" IEEE CG&A. Vol.11 No. 6, Pag. 16-17. 1991

[6] Pang Alex T., "Line-Drawing Algorithms for Parallel Machines" IEEE C.G.&A. Pag. 54-59 Sept. 1990.

[7] Mollá R., Quirós R., Vivó R. "Fixed Point Digital Differential Analyzer" Proceedings of Compugraphics´92. Pag. 1-5. 14-17 Dec. 1992