ABSTRACT
The ePUMA architecture is a novel parallel architecture being developed as a platform for low-power computing, typically for embedded or hand-held devices. As part of the exploration of the platform, we have implemented the Euclidean Distance Transform. We outline the ePUMA architecture and describe how the algorithm was implemented.

KEYWORDS
Euclidean distance transform parallel hardware SIMD.

1. INTRODUCTION
The ePUMA (embedded Parallel DSP with Unique Memory Architecture) is a parallel architecture in development. It exists as a cycle-accurate simulator running assembly level programs. It is primarily been developed for embedded systems, combining low power consumption with significant computing power.

As part of the development of the architecture, relevant algorithms are implemented, with the dual goal to prove usability and/or find desirable improvements to the design. The main tracks for such explorations are two, communication infrastructures and media. The latter includes video coding, graphics and image processing. The present work investigates an ePUMA implementation of the Euclidean Distance Transform.

2. EUCLIDEAN DISTANCE TRANSFORM ALGORITHMS
A Distance Transform (DT) is a well known operation proposed by Rosenfeld & Pfaltz (1966, 1968). Given a binary image, it produces the distance to the nearest feature pixel for every pixel. The original operation was based on scalar values. These scalar DTs are extensively researched, e.g. by Borgefors (1986).

The Euclidean Distance Transform (EDT) by Danielsson (1980) uses vectors. The sequential EDT occasionally produces insignificant errors. Error-free algorithms have been developed by Rutowitz (1990), Ragnemalm (1990), Paglieroni (1992), Saito & Toriwaki (1994), Eggers (1998) and Lucet (2009).

EDT algorithms can be divided into different classes, including raster-scanning algorithms (Danielsson 1980; Ragnemalm 1989), ordered propagation algorithms (Ragnemalm 1990; Eggers 1998), independent row/column processing algorithms (Paglieroni 1992; Saito & Toriwaki 1994), Minkowski operations (Shih and Mitchell 1992) and Voronoi-based algorithms (Rutowitz 1990; Schouten & van den Broek 2004).

Concerning parallelism, the earliest parallel algorithms were proposed by Danielsson (1980) and Yamada (1984). These algorithms process all pixels in every iteration, wasting computation in unchanging areas.

As shown by Ragnemalm (1989), raster scanning algorithms, including scalar ones, are easy to parallelize. Schneider et. al. (2009) implemented this algorithm on a GPU. Furthermore, other parallel GPU computing algorithms have been presented by Rong et. al. (2006) and Cao et. al. (2010).
3. THE EPUMA ARCHITECTURE

The ePUMA architecture (Liu 2008) is a master-SIMD architecture in development. Its simulator, silicon cost estimation and power cost estimation exist on which exploration work is performed.

In ePUMA, a master processor controls 8 SIMD processors (Figure 1), each with 8 16-bit data paths for most processing. The SIMD processors and the master are connected by a star-ring DMA network. Computations are pipelined in 12-13 or 9-10 steps for different kinds of operations.

Each SIMD processor holds three Local Vector Memory banks (LVM) of 80 kB each (5k vectors with 16 bytes per vector) as fast local memory. Typical LVM handling is illustrated in Figure 2 (left). Two LVMs take turns, one providing input data and space for output, and the other being used for DMA transfers to and from the host, while a third (m0 in the figure) holds persistent data, accumulated output or static input data over many iterations. This scheme overlaps processing with input and output, hiding the cost of memory access.

An alternative approach is shown in the middle of Figure 2, with two LVM’s switching roles, is to use a cyclic scheme for all three. For such a case, there should be no persistent data. This is highly suitable for our problem at hand since the output from each iteration (one row/column) is the input to the next.

Image processing is highly relevant for the ePUMA project. Ongoing work includes linear operators (such as Sobel operators), nonlinear operators (e.g. median and max) as well as vital components for video codecs (Liu 2008). The task of rendering graphics has also been investigated (Ragnemalm and Liu 2010). However, no prior research on distance transformations has been performed for the platform.

4. RASTER SCANNING EDT FOR SIMD PROCESSING ON THE EPUMA

We have chosen to study raster scanning algorithms, being straightforward, simple, and relatively easy to convert to parallel form. We use the four-scan algorithm proposed by Ragnemalm (1989), more recently implemented on GPUs by Schneider et. al. (2009). We focus on the unsigned 4-scan, 8-neighbor version. Every pixel carries three values, the squared distance \( d^2 \) and a two-component vector \( \mathbf{v} \) (\( \mathbf{v}_x \) and \( \mathbf{v}_y \)). As illustrated by Figure 3, each mask is applied as a separate scan, row by row (top and bottom masks) or column by column (left and right masks). For a sequential algorithm, this means a double "for" loop for each, accessing three neighbors, calculating a candidate value for \( d^2 \) and comparing it to the \( d^2 \) of the center pixel.
Figure 3. SIMD compatible 4-scan EDT (unsigned version).

All applications of the mask are independent within a row (column) with arbitrary order of operation. This makes the algorithm suitable for SIMD implementation. Each row must be processed in sequence, but the computations within one row can be made in parallel. In pseudo code, a mask is applied like this:

```plaintext
for all pixels p in a row
  for each neighbor q on the next row
    d^2_{candidate} := d^2_p + change due to offset
    if d^2_{candidate} < d^2_p then
      d^2_p := d^2_{candidate}; v_p := v_q + offset
```

The \( d^2 \) values are calculated incrementally using only additions and shifts, an acceleration trick rarely noted about distance transforms, useful for low-level implementations such as ours. For example:

\[
||(x+1, y+1)||^2 = (x+1)^2 + (y+1)^2 = x^2 + 2x + 1 + y^2 + 2y + 1 = ||(x, y)||^2 + 2x + 2y + 2
\]

For the ePUMA, we must use local memory, and use all the eight SIMD units. To do this, we upload two rows split into sections, 1/8 row plus one pixel padding, into LVMs. The previous row, which data is read from, should be in one LVM, and the destination, in another, while the third is busy writing out the result of the previous iteration and reading in the next row, which suits the rotating LVM usage of Figure 2 (middle).

Note that after processing a row, the edge pixels must be communicated to the neighbor SIMD. This can be done efficiently with a messaging system built into the system, handling small messages over the ring bus. The ePUMA is programmed in assembly code. The following instructions are relevant for our purposes:

- **CMPW**, Compare Words
- **COPY**, copy data and **COPY.ULT**, Vector COPY if Unsigned Less Than
- **LSLW**, Logical Shift Left Word (for 2^a)
- **ADDW**, ADD Word and **ADDW.ULT**, ADD Word if Unsigned Less Than

All operations operate on 8 16-bit integers in parallel, which is what we use in our implementation. Thus, in order to process 8 pixels with a neighbor, we need to (with appropriate addressing):

- **ADDW** and **LSLW** to get \( d^2 + v.x*2 + v.y*2 + 2 \) or \( d^2 + v.y*2 + 1 \)
- **CMPW** between \( d^2 \) and the \( d^2 \) of the center pixel.
- **COPY.ULT** and **ADDW.ULT** \( d^2, v.x+1 \) and \( v.y+1 \) to the center pixel.

Operations need to be rearranged to avoid pipeline stall. We have performed such rearranging below to get realistic performance. The full code for the algorithm (optimized, for 128 pixels at a time) is as follows:

```assembly
// load 8 d2-values for comparison
copy vr0 m0[ar0+1].v
repeat 16 REPEAT_END // repeat 16 times

// Left pixels
lslwq vacr0 m1[ar0+1].v 1  // 2x
  2 * addwq vacr0 m1[ar1].v vacr0 // 2x + 2y
addwq vacr0 m1[ar2].v vacr0 // d2 + 2x + 2y
addwq vr1/vacr0 2 vacr0 // d2 + 2x + 2y + 2
cmpwq vr0 vacr0 // compare with d2 + 2x + 2y + 2 -> first flags

// Start with middle pixels
lslwq vacr0 m1[ar0+1].v 1  // 2x
  addwq vacr0 m1[ar2+1].v vacr0 // d2 + 2x
addwq vr1/vacr0 1 vacr0 // d2 + 2x + 1
cmpwq vr0 vacr0 // compare with d2 + 2x + 1 -> second flags

// First flags done, update: Conditionally copy/add if ult (unsigned less than)
copy.ult m0[ar2].v vr1
```
addwq.ult m0[ar0].v m1[ar1]+1 1
addwq.ult m0[ar1].v m1[ar0]+1 1

// Right pixels
laeq vacr0 m0[ar0+2].v 1  // 2x
2 * addwq vacr0 m1[ar1+2].v vacr0  // 2x + 2y
addwq vr1/vacr0 2 vacr0  // d2 + 2x + 2y + 2
cmpwq vr0 vacr0  // compare with d2 + 2x + 2y + 2 -> third flags

// Second flags done -> Update
copy.ult m0[ar2].v vr1
addwq.ult m0[ar1].v m1[ar0+1].v
addwq.ult m0[ar1].v m1[ar0+1].v 1
addwq.ult m0[ar0+9].v m1[ar0+9].v 1
addwq.ult m0[ar1+8].v m1[ar1+8].v 1

// Third flags done -> Update
addwq.ult m0[ar0+8].v vr1
addwq.ult m0[ar0+8].v m1[ar0+8].v 1
addwq.ult m0[ar1+8].v m1[ar1+8].v 1

REPEAT_END:
10 * nop

This sums to 26 cycles per loop or a total of 16*26+10 = 426 cycles. This will process 128 pixels. We will need four similar passes over the image for a complete transform, plus the cost for a square root of 17 cycles for 8 pixels (Tomasson 2010) or 272 cycles for 128 pixels. The cycle cost per pixel will thus be (426 * 4 + 272) / 128 = 15.4 cycles per pixel! To this should be added a slight overhead for the messaging between row segments and transposing the image once in order to access data row by row even in the vertical passes. The former is negligible, while the latter can be done very efficiently, at data transfer cost with full bandwidth usage, 8/3 cycles per pixel.

The total processing cost is in good balance with the time consumption for memory transfer. This shows both that the algorithm fits the architecture and that it can be implemented with very good performance.

A limitation in our implementation is the use of 16-bit arithmetics. For the vectors (vx and vy) this is not a problem, but more so for the squared distance value (d2). However, the ePUMA supports saturation arithmetics, where overflowing values will simply result in the maximum value (65535 for unsigned integers). The ePUMA can also work in 32-bit mode, using 4 lanes instead of 8.

5. RESULTS AND FUTURE WORK

Making this implementation, we encountered limitations and thereby desirable enhancements to the design, which is the primary result of this work. In this section, we outline the findings and suggested improvements.

The bit vector field set by e.g. the VCMP instruction is somewhat limited. Currently, it will set a single bit for a scalar comparison, and all 8 bits by a vector comparison. We found that this does not properly support mixing of 16- and 32-bit arithmetics. A particularly important addition is the ability to allow the combination of two 4-element vector comparisons to be applied on 8-element vector copying. Then we could work with the d2 array in 32 bits per element but 16-bits for vx an vy, allowing 32-bit precision where needed but still enabling full 8-element operations for all other computations. The cost of those changes are expected to be minor while the value for the possibility to optimize other algorithms is significant.

Also, our implementation indicates very high performance even for a low power ePUMA system.

6. CONCLUSION

We have presented a parallel implementation of the Euclidean Distance Transform for the ePUMA architecture. The algorithm we base our work on is a fairly well-known variant of the Euclidean Distance Transform, but had to be adapted for the ePUMA memory system in order to be efficient. The algorithm is the first inherently sequential image processing algorithm so far to be implemented on this platform.

The algorithm will produce one pixel every 2.3 cycles. For a 500 MHz ePUMA this indicates a processing time of 1.2 ms for a 512x512 image, indicating that more operations as well as larger data sets can be processed by the ePUMA even in low-power configurations.
Based on this work, we have discovered some desirable enhancements for the architecture, for more powerful conditional operations and better mixing of 16- and 32-bit operations.

ACKNOWLEDGEMENT

The ePUMA project is funded by SSF, the Swedish board for strategic research.

REFERENCES

Tomasson, O., 2010, Impl. of Elementary Functions for a Fixed Point SIMD DSP Coprocessor, Linköping University.