COMS 4995

ACC

Emily Sillars

12/22/21

Abstract

While the invention of ASCII art could be attributed in part to necessity[1], the humor and fastidiousness of computer programmers seems to be the larger culprit. One of the earliest examples of ASCII graphics is Leon Harmon's and Kenneth Knowlton's image of choreographer Deborah Hay, created for a prank in which they hung the picture in a coworker's office at Bell Labs[3]. Nowadays, the genre of ASCII art continues to spark humorous, creative projects and to evolve, most notably expanding to include the creation of ASCII animation. Setting humor aside, the technical problem of converting an image to an ASCII representation, or the more complex problem of converting an animation to ASCII is not only interesting, but lends itself well to a parallel implementation. The AAC program (ASCII Animation Converter) takes a group of images and converts them each to an ASCII graphics equivalent. The following sections in this report aim to describe the image to ASCII conversion process, the sequential and parallel algorithms implemented, and performance results.

1 Converting an Image to ASCII

A digital image is represented as a grid of pixels. For simplicity, let us first consider gray scale images. A gray scale image is represented as a 2 dimensional array of pixels, one dimension for the width of the image and one dimension for the length. Each pixel contains a numeric value representing brightness. With a byte representation of a pixel, 256 unique levels of gray can be represented: 0 is black, 255 is white, and the values 1-254 in between are levels of gray increasing in brightness. Taking the definition of an image as a grid of pixels, and a pixel as a byte-sized a numeric value, image to ASCII conversion can be modeled with a simple map from pixels to characters.

While a bijective map (assigning a unique ASCII character to each of the 256 levels of gray) is possible, surjective maps using "character ramps" representing a smaller range of grays are often used. Paul Bourke's webpage "Character representation of grey scale images" [4] outlines an example character ramp for 70 levels of gray:

```
"$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\|()1{}[]?-_+~<>i!lI;:,"^`'. "
```

Using a 70 character ramp, image to ASCII conversion can be modeled as a surjective map from pixels to characters, with the conversion function being

```
f(x)= x % 70
```

Converting a color image to an ASCII representation can then be separated into two steps: 1) converting the color image to gray scale 2) converting from the gray scale representation to ASCII using the process outlined above.

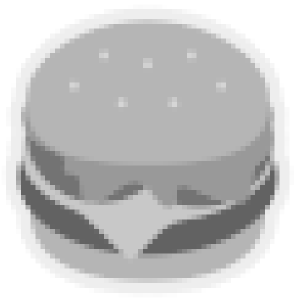

Figure 1: A 64 x 64 pixel gray scale image

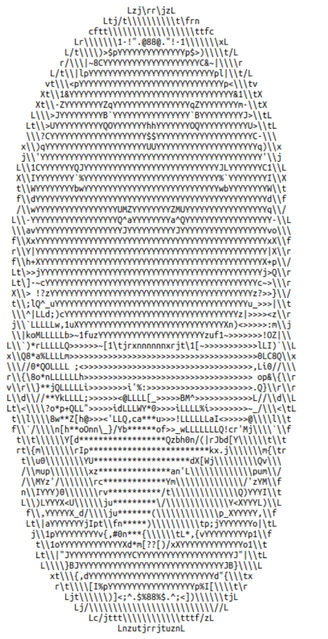

Figure 2: A 64 x 64 character ASCII image equivalent

A digital color image in RGB format can be represented as a 3 dimensional array of pixels, one dimension for width, one dimension for height, and one dimension for the "depth" or color channels. One can visualize a color image then as a 3 dimensional box with a width of image width, a height of image height, and a depth of three units for the red, blue, and green color channels. This third dimension can be flattened by performing a function which takes a pixel's red, green, and blue values as input, and outputs a single gray value. Different definitions of this function exist, but a simple one used by Lescurel on stack exchange.com [2] is

```
f(R,G,B) = 0.2989 * R + 0.5870 * G + 0.1140 * B
```

where f(R,G,B) is the corresponding gray value.

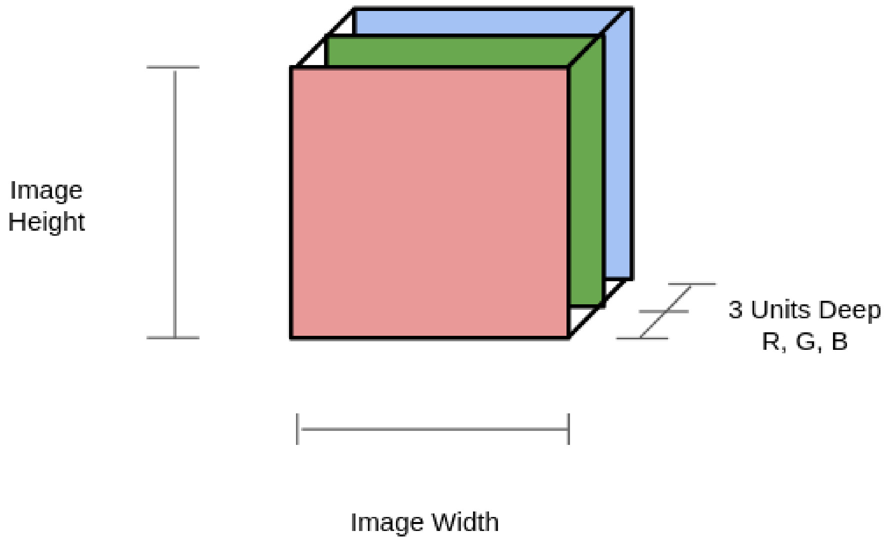

Figure 3: A 3D representation of an the RGB digital image

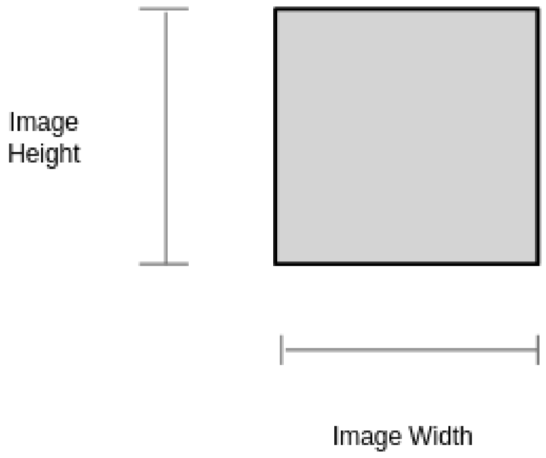

Figure 4: A 2D (flattened) grayscale equivalent of RGB image

2 Sequential Algorithm

For a single image, the steps in the algorithm are

1. read in the file
2. convert the image

3. print out the result

For a group of n images, this algorithm is repeated once for each of the n images .

For the conversion step, I chose to use REPA, which is a library well suited for parallel array computation. The following code snippet using REPA displays the color to gray scale and character ramp functions in use in the conversion process. Using computeS ensures the computation is performed sequentially.

```
surjectionS :: Array D DIM3 Word8 -> Array U DIM2 Char
surjectionS pixels = computeS a
 where
  a                    = fromFunction (Z :. height :. width) toGray
  (height : width : _) = reverse $ listOfShape $ extent pixels
  toGray               = \(Z :. i :. j) ->
    let r = fromIntegral $ pixels ! (Z :. i :. j :. 0)
    in  let g = fromIntegral $ pixels ! (Z :. i :. j :. 1)
        in  let b = fromIntegral $ pixels ! (Z :. i :. j :. 2)
            in  ramp ‘BC.index‘ (gray (r, g, b) ‘mod‘ 70)
```

3 Parallel Algorithms

Parallelizing the conversion algorithm for a single image is as simple as changing computeS to computeP.

For converting a group of images (frames in an animation), I intended to read in and convert each image in parallel with the par monad. The first challenge I ran into was the par monad' library's inability to perform IO operations. It is impossible to parMap over a set of functions that take in or return objects in the IO monad. To bypass this problem, I started reading in each of the image files as a lazy bytestring, and then fed the unwrapped lazy bytestrings to the functions processed by parMap. The remaining chunks of each file could then be read in in parallel by converting from a lazy to a strict bytestring.

After implementing such an algorithm using parmap, I discovered that REPA does not support nested parallelism. According to REPA's documentation, "you cannot map a parallel worker function across an array and then call computeP to evaluate it"[5]. Running parmap over my set of image-to-asci REPA computations forcesdthem to run sequentially. Adapting to this new information, I created two alternative parallelized algorithms nicknamed parMap + REPA and parMap + parMap.

parMap + REPA:

1. Read in files in parallel (parMap)

2. Convert one file after another, with image to ASCII step parallelized (REPA)

3. Print out the results

parMap + parMap:

1. Read in files in parallel (parMap)

2. Convert each file in parallel, with image to ASCII step parallelized (parMap)

3. Print out the results

4 Performance Results

I achieved approximately a 1.82 x speedup using parMap + REPA. parMap + parMap ran exceptionally slowly, and the sequential version consumed the least memory.

Data	Sequential	parMAp + REPA	parMAp + parMap
Total Time	1.31s	0.72s	4.56s
Mutator Time	1.16s	0.70s	2.86s
GC Time	0.15s	0.03s	1.70s
Max Heap Size	29.0MiB	104.0MiB	470.0MiB

Table 1: Threadscope Data

I believe parMap+parMap ran slowly because there was too much overhead computing an ivar for each pixel of the image. Next steps would be to break the image into groups of pixels, and then parMap over these groups to see if reduced granularity improves performance. Converting from lazy bytestrings to strict bytestrings, an expensive operation, likely contributed to the increased memory consumption of the two parallel implemetations.

5 Conclusion

While a modest speedup was achieved , REPA's inability to work in nested parallel settings limited its performance improvement. For the parMap + REPA version, I am extremely interested to see if it would be possible to replace the top level parMap with another REPA computation. Instead of running parMap over a list of images, I could treat the list as a 2D REPA instead , effectively turning the framess-to-ascii conversion for an animation into one giant parallel REPA computation. This proposed method could be explored in future investigations.

References

1. https://www.geeksforgeeks.org/converting-image-ascii-image-python/
2. https://codereview.stackexchange.com/questions/263823/haskell-convert-an-image-to-ascii-art
3. https://collections.vam.ac.uk/item/O239963/studies-in-perception-i-print-harmon-leon/
4. http://paulbourke.net/dataformats/asciiart/
5. https://hackage.haskell.org/package/repa-3.4.1.4/docs/Data-Array-Repa.html

Appendix: Program Listing

A app/Main.hs

```
module Main where
import           Control.Monad.Par              ( parMap
                                                , runPar
                                                )
```

```
import qualified Data.ByteString.Lazy          as BL
import           Data.List                      ( sort )
import           Lib                            ( convertIVar
                                                , convertRepa
                                                , convertS
                                                , readLazyImg
                                                )
import           System.Directory               ( listDirectory )
import           System.Environment             ( getArgs
                                                , getProgName
                                                )
import           System.Exit                    ( die )

main :: IO ()
main = do
    args <- getArgs
    case args of
        [dir] -> do -- sequential
            files <- listDirectory dir
            if null $ sort files
                then
                    die
                    $  "Error: "
                    ++ dir
                    ++ " must contain at least one png file"
                else do
                    let converted =
                            (\x -> do
                                    convertS $ concat [dir, "/", x]
                                )
                                <$> files
                    mapM_
                        (\x ->
                            (do
                                str <- x
                                putStrLn str
                            )
                        )
                        converted
        [dir, "-repa-par-read"] -> do -- parallel read
            files <- listDirectory dir
            if null $ sort files
                then
                    die
                    $  "Error: "
                    ++ dir
                    ++ " must contain at least one png file"
                else do
```

```
                      -- read in part of each file sequentially
                    lazyFrames <-
                        rewrap
                        $   (\x -> BL.readFile $ concat [dir, "/", x])
                        <$> files
                    let frames    = runPar $ readLazyImg `parMap` lazyFrames
                    -- convert files sequentially
                    -- with each image's pixel conversions in parallel
                    let converted = convertRepa <$> frames
                     -- print result sequentially
                    mapM_ putStrLn converted
        [dir, "-ivars"] -> do
            files <- listDirectory dir
            if null $ sort files
                then
                    die
                    $  "Error: "
                    ++ dir
                    ++ " must contain at least one png file"
                else do
                      -- read in part of each file sequentially
                    lazyFrames <-
                        rewrap
                        $   (\x -> BL.readFile $ concat [dir, "/", x])
                        <$> files
                    -- convert each file in parallel
                    let converted = runPar $ convertIVar `parMap` lazyFrames
                    -- print result sequentially
                    mapM_ putStrLn converted
        _ -> do
            pn <- getProgName
            die $ "Usage: " ++ pn ++ " <directory path> <optional p flag>"

rewrap :: [IO BL.ByteString] -> IO [BL.ByteString]
rewrap fls = foldr combine (return [] :: (IO [BL.ByteString])) fls
  where
    combine :: IO BL.ByteString -> IO [BL.ByteString] -> IO [BL.ByteString]
    combine x acc = do
        thing <- x
        list  <- acc
        return (thing : list)
```

B src/Lib.hs

```
module Lib
  ( convertS
  , convertIVar
  , convertRepa
  , readLazyImg
  ) where
import           Codec.Picture                 as J
import           Codec.Picture.Repa            as R
                                                ( Img(imgData)
                                                , RGB
                                                , convertImage
                                                )
import           Control.DeepSeq                ( NFData )
import           Control.Monad                  ( join )
import           Control.Monad.Par              ( parMap
                                                , runPar
                                                )
import           Control.Parallel.Strategies    ( parList
                                                , rdeepseq
                                                , runEval
                                                , using
                                                )
import           Data.Array.Repa               as A
                                         hiding ( (++) )
import qualified Data.ByteString               as B
import qualified Data.ByteString.Char8         as BC
import qualified Data.ByteString.Lazy          as BL
import           Data.Functor.Identity         as F
import qualified Data.List                     as L
import           Data.List.Split                ( chunksOf )
import           Data.Text                      ( pack )
import           Data.Text.Encoding            as TSE
import           Data.Text.Internal.Fusion.Size ( charSize )
import           Data.Typeable                  ( typeOf )
import           Data.Vector.Storable          as V
                                                ( toList )
import           Data.Word                      ( Word8 )
import           GHC.ExecutionStack             ( Location(functionName) )
import           GHC.RTS.Flags                  ( TickyFlags(tickyFile) )
import           Text.Printf                    ( IsChar(toChar) )


-- Paul Bourke’s 70 levels of gray character ramp
ramp :: B.ByteString
ramp = TSE.encodeUtf8 $ pack $ reverse
  "$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\\|()1{}[]?-_+~<>i!lI;:,\"^‘\’. "

-- gray = f(R,G,B) = 0.2989 * R + 0.5870 * G + 0.1140 * B
gray :: (Double, Double, Double) -> Int
gray (r, g, b) = round $ 0.2989 * r + 0.5870 * g + 0.1140 * b
```

```haskell
-- | Sequential Version

-- convert an image to ascii and return as string
-- helper for sequential implementation
convertS :: FilePath -> IO String
convertS png = do
  img <- J.readImage png
  case img of
    (Right v) -> return bi
     where
      imgRGB    = convertRGB8 v
      w         = imageWidth imgRGB
      imgRepa   = R.convertImage imgRGB :: Img RGB
      bi        = L.intercalate "\n" $ chunksOf w $ A.toList imgAsText
      imgAsText = surjectionS $ imgData imgRepa
    (Left err) -> return $ "Read Error: " ++ err

-- sequential image to ascii conversion using REPA
surjectionS :: Array D DIM3 Word8 -> Array U DIM2 Char
surjectionS pixels = computeS a -- :: Array D DIM2 Char -- computeS a
 where
  (height : width : _) = reverse $ listOfShape $ extent pixels
  a                    = fromFunction (Z :. height :. width) toGray
  toGray               = \(Z :. i :. j) ->
    let r = fromIntegral $ pixels ! (Z :. i :. j :. 0)
    in  let g = fromIntegral $ pixels ! (Z :. i :. j :. 1)
        in  let b = fromIntegral $ pixels ! (Z :. i :. j :. 2)
            in  ramp `BC.index` (gray (r, g, b) `mod` 70)

-- | parMap + REPA version

-- convert image to ascii array, then convert to string
-- top level wrapper for surjectionP
convertRepa :: Either String (Image PixelRGB8) -> String
convertRepa (Left  err) = err
convertRepa (Right img) = L.intercalate "\n" $ chunksOf w $ A.toList imgAsText
 where
  repaImg     = R.convertImage img :: Img RGB
  imgAsText   = surjectionP $ imgData repaImg
  (_ : w : _) = reverse $ listOfShape $ extent $ imgData repaImg

-- parallel image to ascii conversion using REPA
surjectionP :: Array D DIM3 Word8 -> Array U DIM2 Char
surjectionP pixels = runIdentity $ computeP a
 where
  (height : width : _) = reverse $ listOfShape $ extent pixels
  a                    = fromFunction (Z :. height :. width) toGray
  toGray               = \(Z :. i :. j) ->
```

```
    let r = fromIntegral $ pixels ! (Z :. i :. j :. 0)
    in  let g = fromIntegral $ pixels ! (Z :. i :. j :. 1)
        in  let b = fromIntegral $ pixels ! (Z :. i :. j :. 2)
            in  ramp `BC.index` (gray (r, g, b) `mod` 70)

-- | parMap + parMap version
-- convert image to ascii array, then convert to string
-- top level wrapper for surjectionIVar
convertIVar :: BL.ByteString -> String
convertIVar png =
  -- finish reading in file (convert from lazy to strict bytestring)
  let img = myReadPng png
  in  case img of
        (Right v) -> imgAsText
         where
          -- convert image to text
          imgAsText = surjectionIVar imgRGB
          -- represent each pixel with three bytes, one for R, G, and B
          imgRGB    = convertRGB8 v
        (Left err) -> "Read Error: " ++ err

-- parallel image to ascii conversion using parmap on gray pixels
surjectionIVar :: Image PixelRGB8 -> String
surjectionIVar img =
 -- separate each row of characters with a newline
                     L.intercalate "\n" $ chunksOf (imageWidth img) chars
 where
   -- convert pixels to ascii characters in parallel
  chars   = runPar $ toChar `parMap` V.toList pixels
  toChar  = \px -> ramp `BC.index` (fromIntegral px `mod` 70)
  pixels  = imageData grayImg
  -- convert color pixels to gray pixels sequentially
  grayImg = pixelMap toGray img
  toGray :: PixelRGB8 -> Pixel8
  toGray (PixelRGB8 r g b) =
    round
      $ 0.2989
      * fromIntegral r
      + 0.5870
      * fromIntegral g
      + 0.1140
      * fromIntegral b

-- | Parallel Reading Helper Functions

-- Convert Lazy ByteString to Image
readLazyImg :: BL.ByteString -> Either String (Image PixelRGB8)
readLazyImg png =
    -- finish reading in file (convert from lazy to strict bytestring)
  let img = myReadPng png
  in  case img of
          -- convert file to Juicy Pixel image
        (Right v  ) -> Right (convertRGB8 v)
        (Left  err) -> Left ("Read Error: " ++ err)

myReadPng :: BL.ByteString -> Either String DynamicImage
myReadPng = myWithImageDecoder J.decodeImage

myWithImageDecoder
  :: (NFData a)
  => (B.ByteString -> Either String a)
  -> BL.ByteString
  -> Either String a
myWithImageDecoder decoder path = decoder $ BL.toStrict path
```