Generating a Fast GPU Median Filter with Haskell

Michal Dobrogost

June 5th, 2014
Overview

- Entire approach is based on a paper by Perrot\cite{1}.
- Thank you Point Grey Research

**Overview**
- GPU Programming
- Median Filter
- Naive Selection
- Forgetful Selection
- DSL Extensions
GPU Programming
3 Slide Crash Course

- 1000’s of concurrent threads
- Memory latency hiding
- Careful
  - Multiple threads share an instruction pointer (SIMT)
  - Memory hierarchy but no cache
  - Static arrays may be far away
- C like languages: CUDA, OpenCL
int sumOfProducts(int k, int* data, int size) {
    if (k == 0) { return 0; }

    int accum = 0;
    for (int i = 0; i < size; i++) {
        accum = accum + k * data[i];
    }
    return accum;
}
void clean(int* data) {
    int cache[3];
    for (int i = 0; i < 3; i++) {
        cache[i] = data[i];
    }
    // Access cache many times
}

void fast(int* data) {
    int c0, c1, c2;
    c0 = data[0];
    c1 = data[1];
    c2 = data[2];
    // Access c0, c1, c2 many times
}
Median Filter

A Road Median

Median Filter

Corrupted Image
Median Filter

Salt and Pepper Noise
Median Filter

Restored Image
Median Filter
How it works

- For each pixel take a 5x5 window

- Linearize pixel intensities into a single collection

- Find the median value
Naive Selection
Sorting Networks
Naive Selection
Domain Specific Language

\[(0,1), (1,2), (2,3), (3,4), (4,5), (0,1), (1,2), (2,3), (3,4), (0,1), (1,2), (2,3), (0,1), (1,2), (0,1)\]
Naive Selection
Leverage Parent Language

```
link :: [Int] -> [(Int, Int)]
link xs = zip xs $ tail xs

bubble :: Int -> [(Int,Int)]
bubble n = concatMap iter [n-1,n-2..1]
    where iter k = link [0..k]
```
Naive Selection

C “compiler”

```haskell
swap :: String -> String -> String
swap x y = "T tmp(" ++ x ++ "); " ++
          x ++ " = " ++ y ++ "; " ++
          y ++ " = tmp;"

printC :: [(Int,Int)] -> String
printC [] = ""
printC ((x,y):xs) = res ++ printC xs
  where x' = "x" ++ show x
        y' = "x" ++ show y
        res = "if (" ++ x' ++ ">" ++ y' ++ ") { " ++
               swap x' y' ++
               " }\n"
```
Naive Selection

Generated C Code

```c
if (x0 > x1) { T tmp(x0); x0 = x1; x1 = tmp; }
if (x1 > x2) { T tmp(x1); x1 = x2; x2 = tmp; }
if (x2 > x3) { T tmp(x2); x2 = x3; x3 = tmp; }
if (x3 > x4) { T tmp(x3); x3 = x4; x4 = tmp; }
if (x4 > x5) { T tmp(x4); x4 = x5; x5 = tmp; }
if (x0 > x1) { T tmp(x0); x0 = x1; x1 = tmp; }
if (x1 > x2) { T tmp(x1); x1 = x2; x2 = tmp; }
if (x2 > x3) { T tmp(x2); x2 = x3; x3 = tmp; }
if (x3 > x4) { T tmp(x3); x3 = x4; x4 = tmp; }
if (x0 > x1) { T tmp(x0); x0 = x1; x1 = tmp; }
if (x1 > x2) { T tmp(x1); x1 = x2; x2 = tmp; }
if (x2 > x3) { T tmp(x2); x2 = x3; x3 = tmp; }
```
printTex :: [(Int,Int)] -> String
printTex comps =
  \documentclass[tikz]{standalone}
  \usepackage{tikz}
  \begin{document}
  \begin{tikzpicture}
  verticalWires ++ connections ++
  \end{tikzpicture}
  \end{document}
where
  verticalWires = ...
  connections = ...
Forgetful Selection

Trace on a 3x3 Window

starting point is 5 = ceil(9/2)

/ 

Data: 0 1 2 3 4 5 6 7 8

| | | | | | | | |
min x x x x x max | | |
| | | | | | |
| | | | | | /---/ |
min x x x max
| | |
| | | |
| | | /-------/ |
min x x max
| | |
| | |
| | | /-----------/ 
min x max
| |
| |
| | |
| | | /-----------/ 
min x max
| |
| |
| | \
median
Forgetful Selection
Row Step — Basic

```haskell
step :: [Int] -> [(Int,Int)]
step ixs = (\x -> x ++ reverse x) $ link ixs

-- step [0..5]
```
Forgetful Selection

Row Step — No Duplicates

-- group :: Eq a => [a] -> [[a]]

noDup :: Eq a => [a] -> [a]
nDup xs = map head $ group xs

step :: [Int] -> [(Int,Int)]
step ixs = noDup $ (\x -> x ++ reverse x) $ link ixs
Forgetful Selection
Row Step — Minimize Dependencies

\[
\text{interleave} :: [a] \rightarrow [a] \rightarrow [a]
\]

\[
\text{interleave} \ [\] \ ys = ys
\]

\[
\text{interleave} \ (x:xs) \ ys = x : \text{interleave'} \ xs \ ys
\]

\[\text{where}\]

\[
\text{interleave'} \ xs \ [\] = xs
\]

\[
\text{interleave'} \ xs \ (y:ys) = y : \text{interleave} \ xs \ ys
\]

\[
\text{step} :: [\text{Int}] \rightarrow [(\text{Int},\text{Int})]
\]

\[
\text{step} \ ixs = \text{noDup} \ (\lambda x \rightarrow \text{interleave} \ x \ (\text{reverse} \ x)) \ \text{link} \ ixs
\]
Forgetful Selection

Full Algorithm

\[
\text{forgetful} :: \text{Int} \rightarrow [(\text{Int}, \text{Int})]
\]
\[
\text{forgetful } n = \text{concatMap step rows}
\]
\[
\text{where } (\text{row0}, \text{rest}) = \text{splitAt } ((n+1) \div 2) [0..n-1]
\]
\[
\text{rows} = \text{zipWith } (\lambda r \ x \rightarrow r ++ [x]) \text{ (tails row0) rest}
\]
Forgetful Selection
Comparison on a 3x3 Window

Forgetful Select (24)

Bubble Sort (36)
DSL Extensions
Late Loads

data DSL = Comp Int Int | Load Int Int

mkComp :: (Int,Int) -> DSL
mkComp (x,y) = Comp x y

genLoads :: (Int,Int) -> Int -> [DSL] -> [DSL]
genLoads (winX,winY) stride instrs = ...

printC :: [DSL] -> String
Conclusion

- Haskell can be extremely effective for throw away scripting
- Suitable datatypes go a long way
- Leverage Haskell to boost your DSL
- Original implementation
  - \(\sim 4\) hours
  - 28 lines
  - *forgetful* was a monster
- Presentation code
  - Time hard to judge – shared with presentation
  - 22 lines
  - *link* is the key ingredient
References

THANK YOU