GTCx

SEOUL | Oct.7, 2016

OPTIMAL MEDIAN FILTER ALGORITHMS FOR IMAGE DATA PROCESSING ON GPU

Hyung-Jin Kim(SAIT)

PRESENTED BY

NVIDIA.

AGENDA

1. Median Filter ?
   - Application
   - Algorithm
2. Implementation on GPU
   - Naïve approach
   - Optimization - Shared Memory, Data reuse, Register,…
3. Variation and Performance

MEDIAN FILTER

Find a median value within the given data set

0.35	0.67	0.23	0.44	0.93	0.90
0.77	0.68	0.15	0.68	0.10	0.85
0.67	0.55	0.28	0.30	0.12	0.75
0.66	0.50	0.31	0.11	0.18	0.78
0.24	0.32	0.21	0.28	0.25	0.66
0.33	0.37	0.24	0.50	0.77	0.62

6x6 pixel image

Sal & Pepper noise

3x3 median filter Proc.

3x3 filtering mask

0.35	0.67	0.23	0.77	0.68	0.15	0.67	0.55	0.28

↓ Ordering numbers

0.15	0.23	0.28	0.35	0.55	0.67	0.67	0.68	0.77

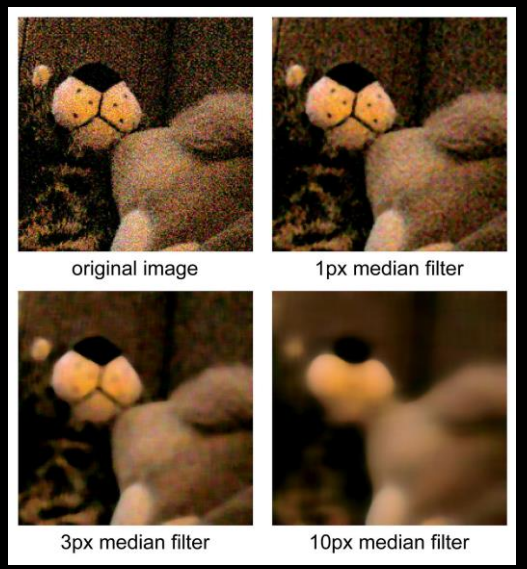
↓ Return median value

0.55

APPLICATION

Noise reduction

Corrupted image recovery



Object edge detection
for Vision processing

MEDIAN FILTER ALGORITHM

Sorting half of the values in fastest way

3x3 filtering mask

0.35	0.67	0.23	0.77	0.68	0.15	0.67	0.55	0.28

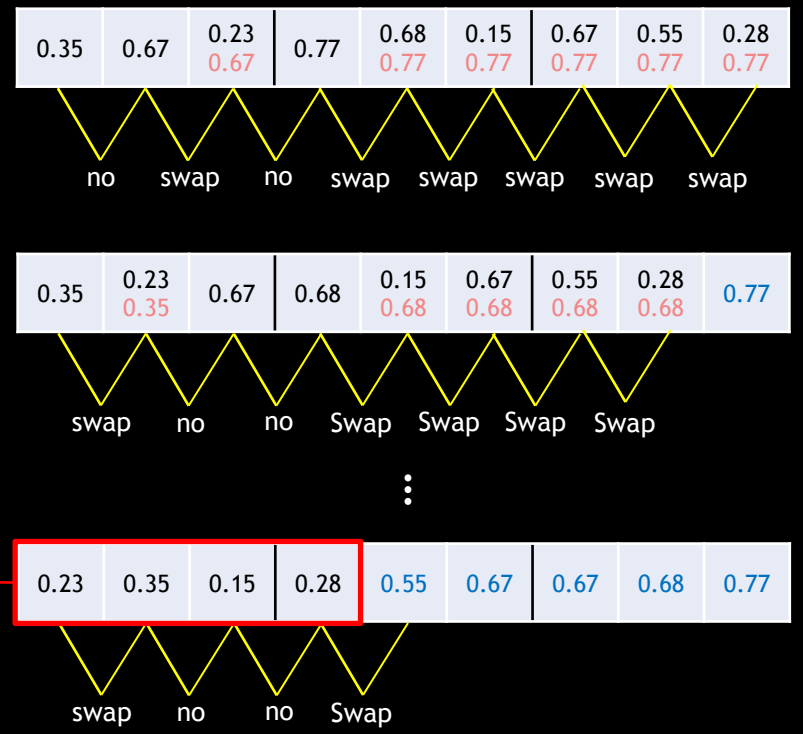
↓ Ordering numbers (How ?)

0.15	0.23	0.28	0.35	0.55	0.67	0.67	0.68	0.77

↓ Return median value

0.55

Find Maximum



For MxN pixels(neglect boundaries)

M x N x 3 x 3 x 4(single) bytes of data access needed

$({}_9C_2 - {}_4C_2)$ x M x N data comparisons needed

Total ${}_9C_2 - {}_4C_2 = 30$ times of comparisons needed

IMPLEMENTATION

CPU version pseudo code

```
void compare(Dest,Src,x,y,mask) {
   int m_sq = mask*mask;
   float *buf = new float[m_sq];
   //Assign masked values to buf
   for(i = 0; i < m_sq; i++)
      buf[i] = Src[(y-1+i/mask)*width+x-1+i%mask];
  //calculate until we get the median value
   for(k = 0; k < (m_sq+1)/2; k++)
      for(l = 0; l < m_sq; l++)
         if ( buf[k] < buf[l] )
            swap(buf[k],buf[l]);
   Dest[y*width+x] = buf[(m_sq+1)/2];
}
// width by height pixels image, 3x3 median filter
int main(…) {
   …
   #pragma omp parallel for collapse(2)
   for(i = edge; i < width-edge; i++)
      for(j = edge; j < height-edge; j++)
         compare(output,input,i,j,3);
   …
```

GPU version pseudo code

```
__global__ void compare(Dest,Src,mask) {
   int x = blockIdx.x*blockDim.x+threadIdx.x
   int y = blockIdx.y*blockDim.y+threadIdx.y
   int m_sq = mask*mask;
   float *buf = new float[m_sq];
   //Assign masked values to buf
   for(i = 0; i < m_sq; i++) {
      buf[i] = Src[(y-1+i/mask)*width+x-1+i%mask];
   }
  //calculate until we get the median value
   for(k = 0; k < (m_sq+1)/2; k++)
      for(l = 0; l < m_sq; l++)
         if ( buf[k] < buf[l] )
            swap(buf[k],buf[l]);
}
// width by height pixels image, 3x3 median filter
int main(…) {
   …
   compare<<<grid,block,…>>>(output,input,3);
   …
```

NAÏVE APPROACH

GPU version pseudo code

```
__global__ void compare(Dest,Src,mask) {
  int x = blockIdx.x*blockDim.x+threadIdx.x
  int y = blockIdx.y*blockDim.y+threadIdx.y
  int m_sq = mask*mask;
  float *buf = new float[m_sq];
  //Assign masked values to buf
  for(i = 0; i < m_sq; i++) {
    buf[i] = Src[(y-1+i/mask)*width+x-1+i%mask];
  }
  //calculate until we get the median value
  for(k = 0; k < (m_sq+1)/2; k++)
    for(l = 0; l < m_sq; l++)
      if ( buf[k] < buf[l] )
        swap(buf[k],buf[l]);
}
// width by height pixels image, 3x3 median filter
int main(...) {
  ...
  compare<<<grid,block,...>>>(output,input,3);
  ...
```

- Naïve approach is just 1-to-1 mapping of CPU version code implementation
  → No GPU hardware consideration, very slow!!!
- GPU global memory access is extremely slow
- Entire image should be accessed for Mask x Mask times but some of them can be shared to next pixel computation
- Data access should be aligned for maximum memory bandwidth

OPTIMIZATION

- Use the fastest memory as buffer : Use Shared memory or Register

```
float *buf = new float[m_sq];
// buffer memory is needed for saving needed numbers : Should be big enough
// Need to be accessed for several time : Low latency, High bandwidth
```

- Maximum memory bandwidth

```
// coalesced memory access
// Texture cache, L2 cache is always useful
```

- Better Algorithm?

```
// Check computation intensive or IO intensive
// Is there a better way to reduce bottle neck?
```

OPTIMIZATION

- Data re-usability

0.35	0.67	0.23	0.44	0.93	0.90
0.77	0.68	0.15	0.68	0.10	0.85
0.67	0.55	0.28	0.30	0.12	0.75
0.66	0.50	0.31	0.11	0.18	0.78
0.24	0.32	0.21	0.28	0.25	0.66
0.33	0.37	0.24	0.50	0.77	0.62

6x6 pixel image

- 6 numbers of violet area can be shared in 1st, 2nd continuous pixels computation
  → Reduce the number of excessive data access
- Shared buffer(not share memory!!!) can reduce the number of needed buffer size
- Share memory buffer
  → Limited memory size, low latency, bank conflict
- Register memory buffer
  → Highly limited number, lowest latency

EXPERIMENT 1

- Buffer : Register, 1D data reuse : shared memory

	0.35	0.67	0.23	0.44	0.93	0.90
Line 1 →	0.77	0.68	0.15	0.68	0.10	0.85
	ThrdID 0	ThrdID 1	ThrdID 2	ThrdID 3	ThrdID 4	ThrdID 5
Line 2 →	0.67	0.55	0.28	0.30	0.12	0.75
Line 3 →	0.66	0.50	0.31	0.11	0.18	0.78
Line 4 →	0.24	0.32	0.21	0.28	0.25	0.66
Line 5 →	0.33	0.37	0.24	0.50	0.77	0.62

6x6 pixel image

- Read the first 3 rows to shared memory: 3x6x4bytes
- Thread block size is set 6 for convenience,
  only 1<threadID<6 of threads are active
- Entire image should be accessed for Mask(=3) times
- Performance limited by CUDA occupancy(depends on Share Mem, Reg)

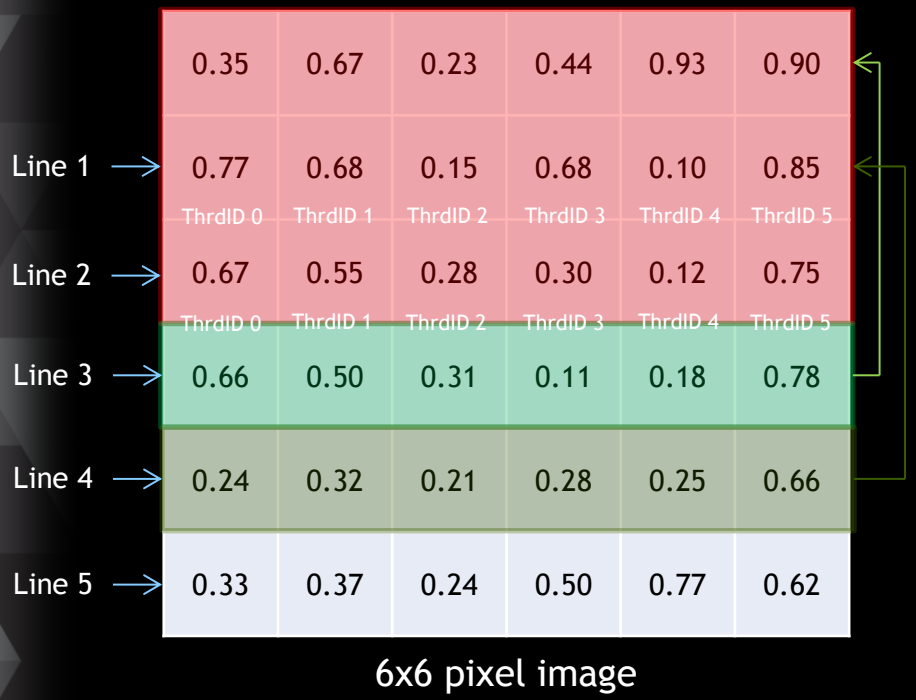
EXPERIMENT 1

GPU version Opt. version 1(pseudo code)

```
#define compare(A,B) \
    if(A<B){   temp=B; B=A; A=temp;   }
#define bID blockIdx.x
#define tID threadIdx.x
__global__ void compare_opt1(Dest,Src,mask) {
  int off = blockIdx.x*blockDim.x+threadIdx.x
  extern __shared__ float data[];
  //assuming Mask = 3
  //read data to SM(shared Mem) for data re-usability
  //line by line reading, thread block size = image width
  if( y_boundary < pixel location < y_boundary)
  data[tID]          = Src[off-width];
  data[tID+width]    = Src[off];                    } //shared memory assignment, read data for Mask times
  data[tID+2*width] = Src[off+width];
  __syncthreads(); //data loaded to SM
  //calculate until we get the median value
  if( x,y_boundary < pixel location < x,y_boundary)
  float buf00 = data[tID-1]; float buf01 = data[tID]; float buf02 = data[tID+1];   } //Assign shared data to register memory
  float buf10 = data[width+tID-1]; float buf01 = data[width+tID]; …
  compare(buf00,buf01); compare(buf00,buf02), …                                   } //Compare the elements for 9C2 - 4C2 times for Line 1 elements
  compare(buf10,buf01); compare(buf10,buf02), …
  …
```

EXPERIMENT 2

- Buffer : Register, 2D data reuse : shared memory



6x6 pixel image

- After finish “Line 1” computation, read “Line 3” data and write it to 1st row of SM

  Reassign bufXX register with new SM

  Calculate “Line 2”
- Again, read “Line 4” data and write it to 2nd row of SM

  Reassign bufXX register with new SM

  Calculate “Line 3”
- Now you have calculated 3 lines with 5 times of data access

  ↔ In previous you have to access the data by 9 times

EXPERIMENT 2

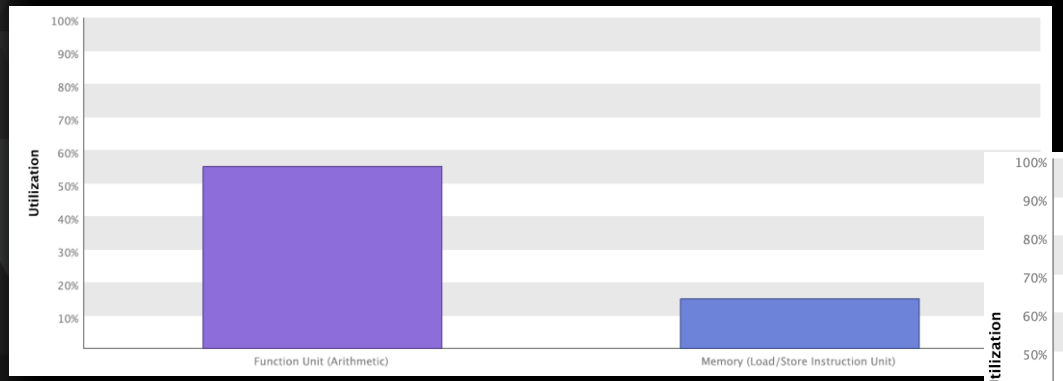
GPU version Opt. version 2(pseudo code)

```
__global__ void compare_opt2(Dest,Src,mask) {
  … // Continue Experiment 1 code,
  if( y_boundary < pixel location < y_boundary)
  data[tID]            = Src[off+2*width];
  __syncthreads(); //data loaded to SM        } //shared memory re-assignment, read Line 3(4th row)
  if( x,y_boundary < pixel location < x,y_boundary)
  float buf00 = data[width+tID-1]; float buf01 = data[width+tID]; float buf02 = data[width+tID+1];   } //Assign new pixel data to register buffer
  float buf10 = data[2*width+tID-1]; … float buf20 = data[tID-1]; …
  compare(buf00,buf01); compare(buf00,buf02), …
  compare(buf10,buf01); compare(buf10,buf02), …   } //Compare the elements for 9C2 - 4C2 times for Line 2 elements
  …
   if( y_boundary < pixel location < y_boundary)
  data[tID+width]    = Src[off+3*width];
  __syncthreads(); //data loaded to SM        } //shared memory re-assignment, read Line 4(5th row)
  //calculate until we get the median value
  if( x,y_boundary < pixel location < x,y_boundary)
  float buf00 = data[2*width+tID-1]; float buf01 = data[2*width+tID]; float buf02 = data[2*width+tID+1];   } //Assign new pixel data to register buffer
  float buf10 = data[tID-1]; … float buf20 = data[width+tID-1]; …
  compare(buf00,buf01); compare(buf00,buf02) …
  compare(buf10,buf01); compare(buf10,buf02) …   } //Compare the elements for 9C2 - 4C2 times for Line 2 elements
  …
      // Three lines of median filter computation with 5 times of row data access
```

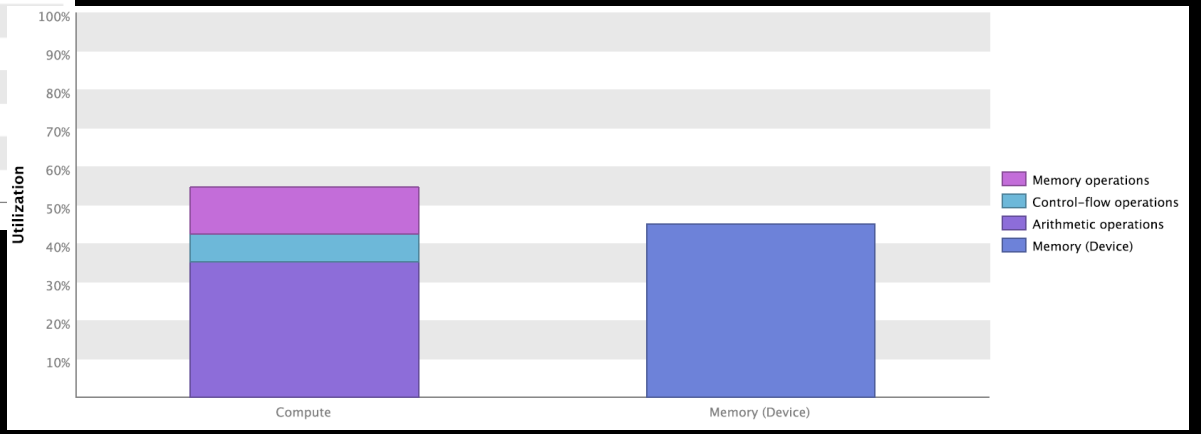
EXP 1,2 PERFORMANCE

- 462(w)x1036(h) pixels image, 2x2 and 5x5 mask are tested on K20Xm GPU

	2 x 2	5 x 5
Exp 1	47.84 usec	868.424 usec
Exp 2	41.761 usec	435.6 usec



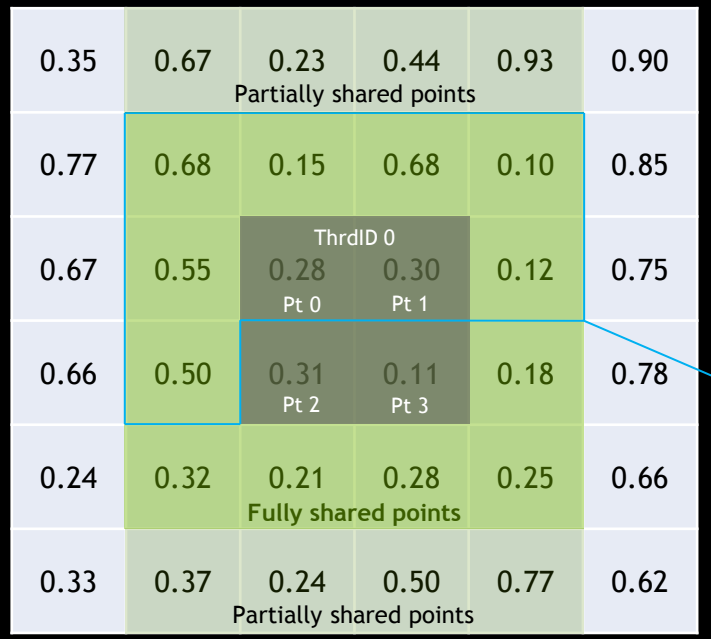
5x5 mask GPU utilization in Exp2



2x2 mask GPU utilization in Exp2

EXPERIMENT 3

- Buffer : Register, 2D data reuse : Register



6x6 pixel image for 5x5 mask

- Shared memory is not that fast in latency
- Too much calculations are needed.

  For 5x5 case, ${}_9C_2 - {}_4C_2 = 234$ arithmetic operations/Pt are needed
- Now let's test a new method for finding the median value

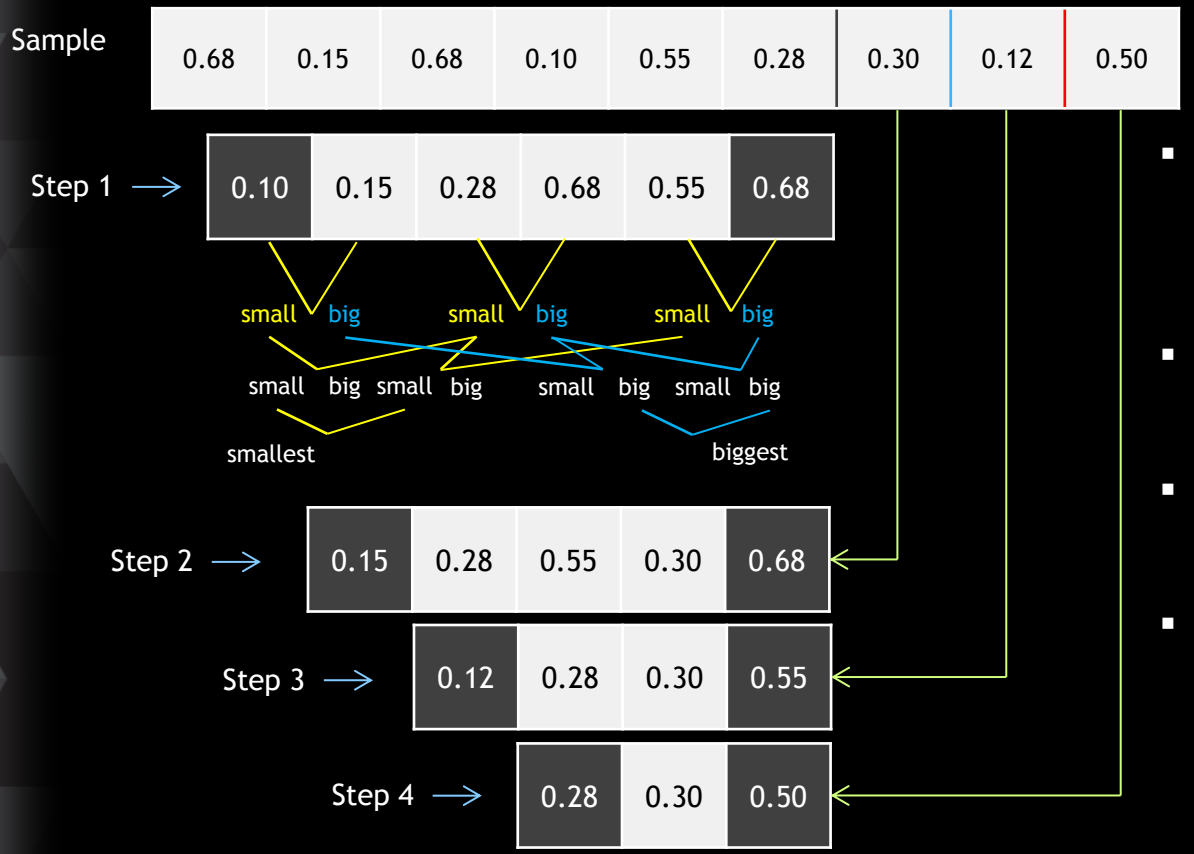
  We will keep removing min, max values among the given number set

  Then for 25 numbers, we can find a median value after 12 steps of sequence

For 9 numbers, we need
4 removing sequence
(∵ 9 - 2*4 < 3)

EXPERIMENT 3

- Modification



- Instead of starting from 9 memory buffers, we can do the same thing with 6 memory buffers and give 1 unassigned element to next eliminating step
- This method has advantage in reducing register requirement and excessive comparing operations on GPU
- For 5x5 mask case, we need 12 steps to get the median value → It corresponds we need 14 registers for buffer
- findMinMax(n elements) can be easily done by divide & conquer method

  For example, findMinMax(14 elements) needs 19 arithmetic operations and findMinMax(13 elements)→18

EXPERIMENT 3

- Modification

0.35	0.67	0.23	0.44	0.93	0.90
		Partially shared points			
0.77	0.68	0.15	0.68	0.10	0.85
0.67	0.55	ThrdID 0 0.28 Pt 0	0.30 Pt 1	0.12	0.75
0.66	0.50	0.31	0.11	0.18	0.78
0.24	0.32	0.21	0.28	0.25	0.66
		Fully shared points			

5x6 pixels for 2 Pt simultaneous calculation using 5x5 mask

- For Pt0 calculation, we have 16 shared pixels, 4 partially shared pixels and 5+5 un-shared pixels
- Therefore, 5+5 unshared pixels will be included in the last 5 steps of two findMinMax sequences, and 4 partially shared points will be included in the steps ahead of unshared sequence
- The ordering information in shared points can be fully reused to Pt0, Pt1, Pt2, Pt3 calculations and the information of partially shared point can be shared Pt0, Pt1 calculations
- Now we can start from findMinMax14, findMinMax13, findMinMax12, …, findMinMax8, 2x findMinMax7, 2x findMinMax6,…, 2x findMinMax3

EXPERIMENT 3

GPU version Opt. version 3(pseudo code)

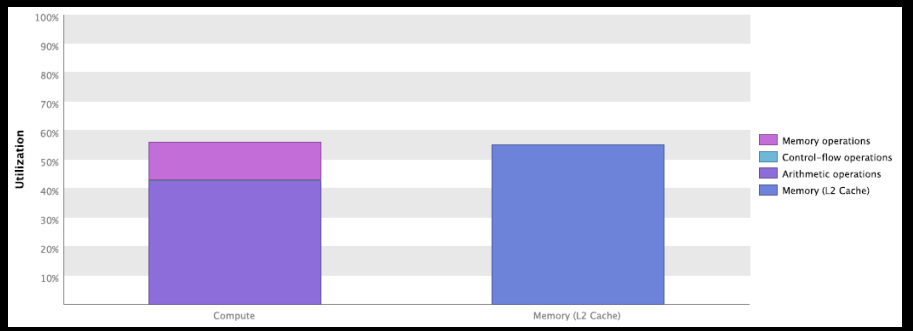
```
__global__ void compare_opt3(Dest,Src,mask) {
  unsigned int mem_off = 2*blockIdx.x*width+2*threadIdx.x;
  float *ref_pt = input+mem_off - width;
  fbuf00 = *(ref_pt-1); fbuf01 = *(ref_pt); fbuf02 = *(ref_pt+1); fbuf03 = *(ref_pt+2); ref_pt += width;
  fbuf04 = *(ref_pt-1); fbuf05 = *(ref_pt); fbuf06 = *(ref_pt+1); fbuf07 = *(ref_pt+2); ref_pt += width;
  …
  maxmin14(fbuf00,fbuf01,…,fbuf13); fbuf13 = *(ref_pt+1);
  maxmin13(fbuf01,fbuf02,…,fbuf13); fbuf13 = *(ref_pt+2);
  maxmin12(fbuf02,fbuf03,…,fbuf13);          // only 10 registers needed for saving fully shared pixels ordering information
  ref_pt = input+mem_off-2*width; pbuf00 = *(ref_pt-1);
  maxmin11(fbuf03,fbuf04,…,fbuf12,pbuf00); pbuf00 = *(ref_pt);
  …                                                                  } // Counted 4 partially shared elements
  maxmin8(fbuf06,fbuf07,…,fbuf12,pbuf00);
  cbuf00 = fbuf07; cbuf01 = fbuf08; … cbuf05 = fbuf12; ubuf0= *(ref_pt-2); ubuf1= *(ref_pt+3); ref_pt+=width;
  maxmin7(fbuf07,…,fbuf12,ubuf0); maxmin7(cbuf00,…,cbuf05,ubuf1); ubuf0= *(ref_pt-2); ubuf1= *(ref_pt+3); ref_pt+=width;
  maxmin6(fbuf08,…,fbuf12,ubuf0); maxmin7(cbuf01,…,cbuf05,ubuf1); ubuf0= *(ref_pt-2); ubuf1= *(ref_pt+3); ref_pt+=width;
  …
  maxmin3(fbuf11,fbuf12,ubuf0); maxmin3(cbuf04,cbuf05,ubuf1);       // Counted 5 un-shared elements
  …
      // We get two of median values for pt_0,pt_1 then you can continue the same sequence for pt_2,pt_3
```

EXP 3 PERFORMANCE

- 462(w)x1036(h) pixels image, 2x2 and 5x5 mask are tested on K20Xm GPU

	2 x 2	5 x 5
Exp 1	47.84 usec	868.424 usec
Exp 2	41.761 usec	435.6 usec
Exp 3	N/A	145.57 usec

- Exp 2 has an advantage in reducing memory access but the number of arithmetic operations is not changed at all.
  Overally for 4 pt computations, we need 4 x (${}_9C_2 - {}_4C_2$) = 936 operations are needed
- But in Exp 3 case, 19+18+16+(15+13+12+10)*2+(9+7+6+4+3)*4 = 269 operations for 4 pt
- Given data size = (36 read+4 write) x ¼ x 458 x 1032 x 4 bytes = 18.9MB
  With maximum bandwidth(~200GB/s), ~94.5 usec is needed → IO and computation is not fully overlapped



5x5 mask GPU utilization in Exp3

CONCLUSION

- In many cases of GPU computation, memory accessing is most important part for optimal performance. Therefore reducing a GPU global memory access and using highest memory performance is the key to GPU code optimization
- In median filter algorithm, floating point operations are not dominant at all, but most of them are branch and memory transaction operations
- Consequently, finding a optimal point between the memory operations and data accesses is the most important factor to get the best performance

GTCx

SEOUL | Oct.7, 2016

Ref - Designing Scientific Applications on GPUs

THANK YOU

JOIN THE CONVERSATION

#GTCxKorea2016

PRESENTED BY

NVIDIA