GPU를 이용한 Video Encoding

VOCEWEB R&D Center
CTO 이재규
jakelee@voceweb.com

Agenda

- Video Encoding Technology Overview
- ME Implementation Using CUDA
- Issues on Parallel Video Encoding
- Q&A

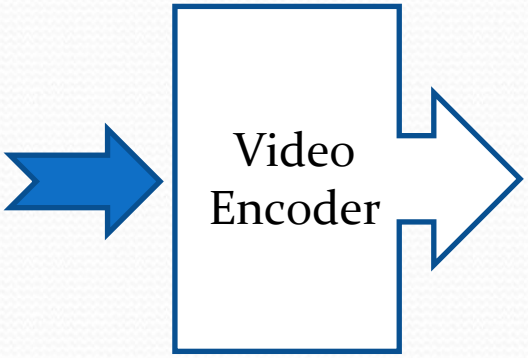
Video Encoding Technology Overview

Video Encoding 왜 중요한가?

- 현대는 Media의 전성 시대
  - HDTV, DMB, HD-DVD, VoD, UCC ...
- 압축하지 않으면 Handling할 수 없는 크기

30fps

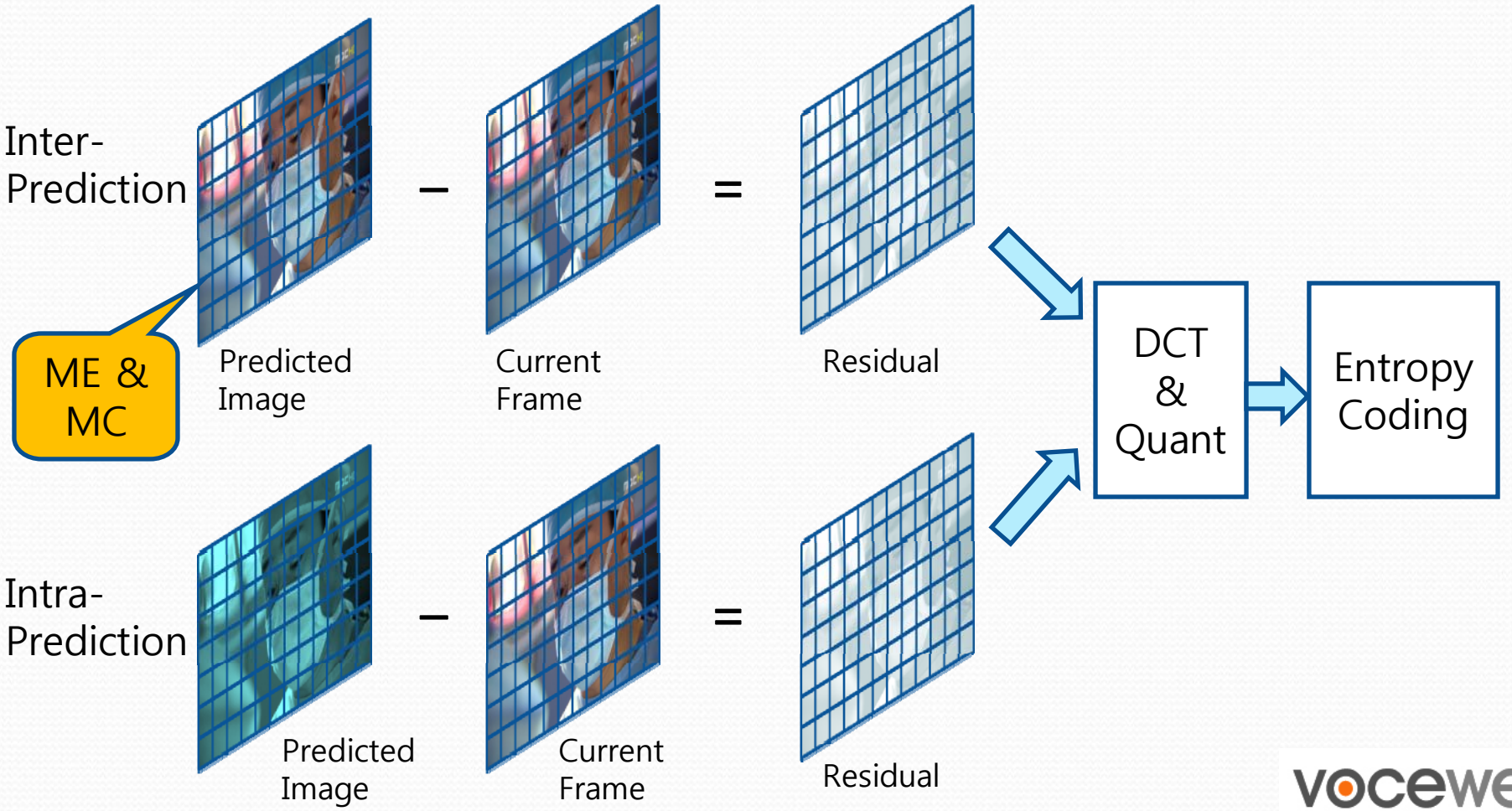
640x352x3x30x8 = 160Mbps



Quality Loss

1Mbps Encoded Stream

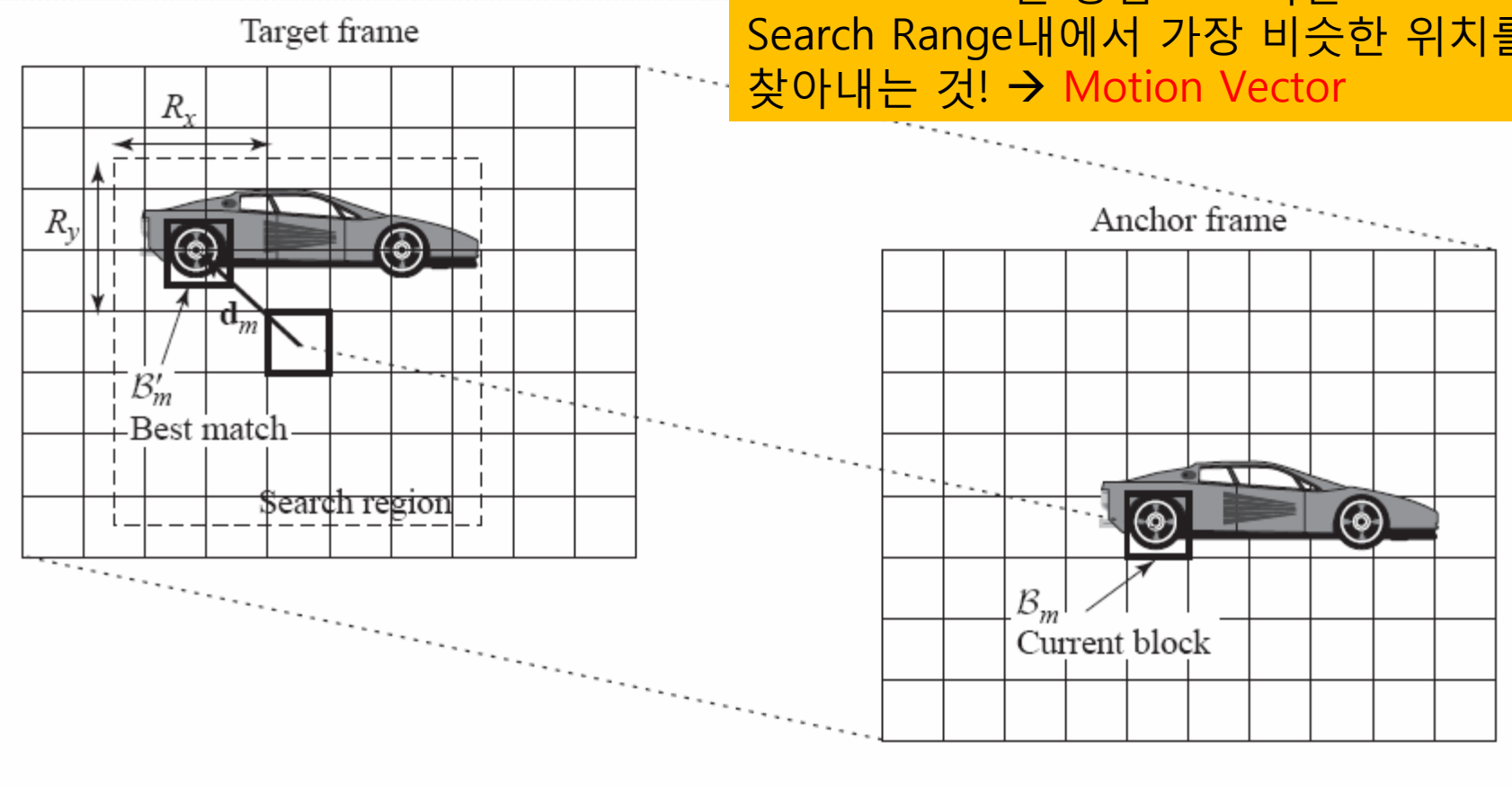
Video Encoding 원리



voceweb

Motion Estimation

Current Block을 중심으로 이전 Frame의 Search Range내에서 가장 비슷한 위치를 찾아내는 것! → Motion Vector

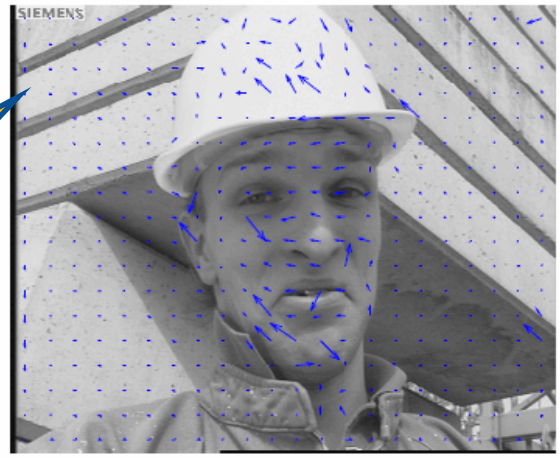


Motion Compensation

Frame 66

Frame 69

Predicted frame69 with MV overlay



Predicted Frame69

ME Result
= MV Fields

Compensated
Image

voceweb

Residual After Compensation

Frame 69

Predicted Frame69

Absolute Difference w/o Motion Compensation

Absolute Difference with Motion Compensation

Residual
만 전송

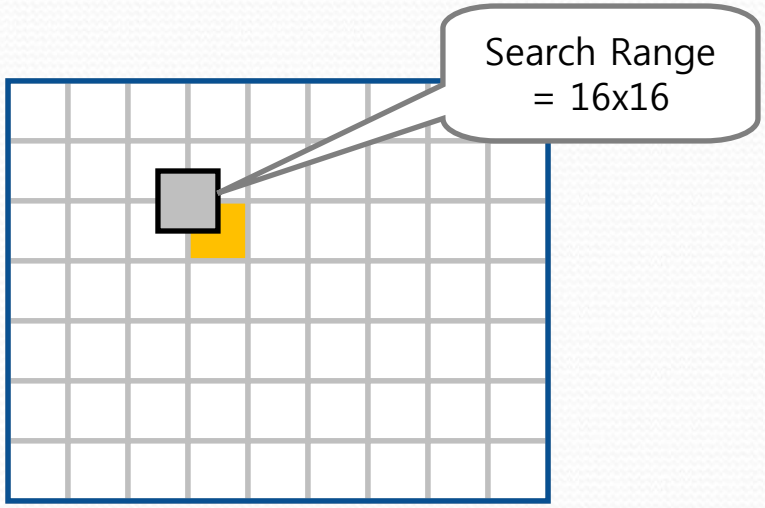
voceweb

ME Implementation Using CUDA

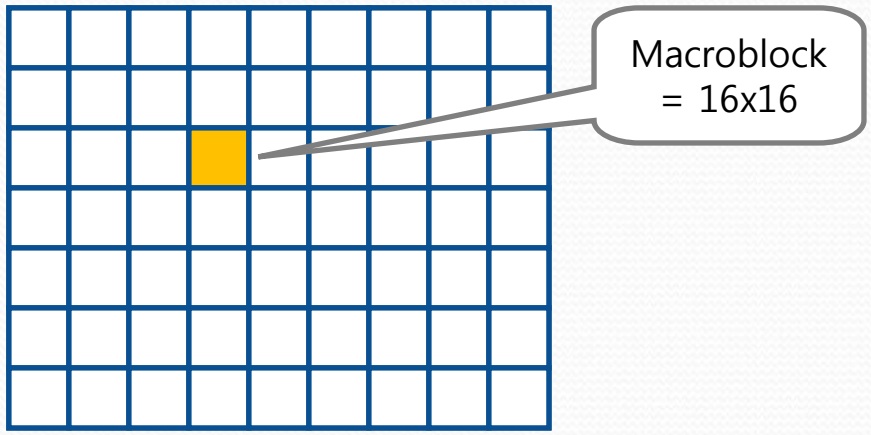
Why Motion Estimation?

- ME는 Video Encoding 속도에서 매우 중요함.
  - Full Search의 경우 80% 비중.
  - Fast Search의 경우 20~30% 비중.
  - Bitrate 대비 Quality는 Full Search가 가장 좋음.
- GPU는 거대한 SIMD Machine
  - Full Search는 매우 많은 독립적인 단순 연산
- GPU를 이용하여 고속/고품질 ME가 가능함.

Example Configuration



Reference Frame (640x480)



Current Frame (640x480)

Distortion = SAD (Sum of Absolute Difference)

SAD 연산 횟수 = (640/16)*(480/16)*(16*16)*(16*16) = 78,643,200

Result = Motion Vector Array (40x30)

CPU = Intel Q6600, RAM = 4GB, Graphic Card = GeForce 8800GTS

CPU Implementation

```
for (int mby = 0; mby < (HEIGHT/MBSIZE); mby++) {
  for (int mbx = 0; mbx < (WIDTH/MBSIZE); mbx++) {
    cx = mbx * MBSIZE;
    cy = mby * MBSIZE;
    minsad = INT_MAX;
    mvx = 0; mvy = 0;
    for (int y = max<int>(0, cy - SR); y < min<int>(HEIGHT, cy + SR); y++) {
      for (int x = max<int>(0, cx - SR); x < min<int>(WIDTH, cx + SR); x++) {
        sad = 0;
        for (int yy = 0; yy < MBSIZE; yy++) {
          for (int xx = 0; xx < MBSIZE; xx++) {
            sad += abs(int(ref[INDEX(x+xx, y+yy)]) - int(cur[INDEX(cx+xx, cy+yy)]));
          }
        }
        if (sad < minsad) {
          mvx = x – cx;  mvy = y - cy;
          minsad = sad;
        }
      }
    }
    mv[(mbx + mby*(WIDTH/MBSIZE))*2] = mvx;
    mv[(mbx + mby*(WIDTH/MBSIZE))*2 + 1] = mvy;
  }
}
```

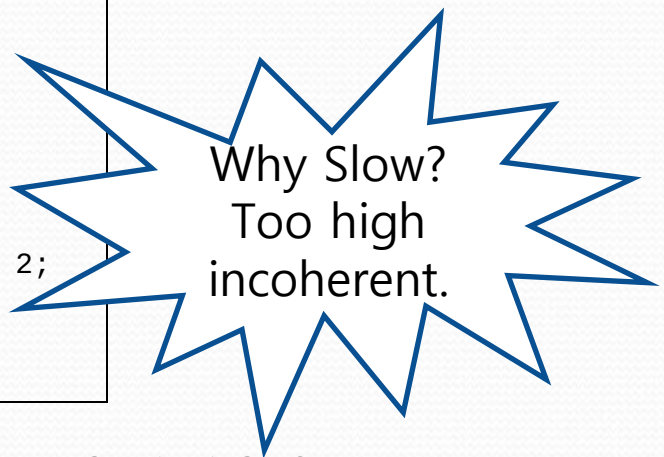
CPU = 94.15 msec

GPU Impl. #1 (Global Memory)

```
int tid = threadIdx.x + threadIdx.y * blockDim.x;
int curleft = blockIdx.x * MBSIZE;
int curtop  = blockIdx.y * MBSIZE;
int refleft = max(curleft - SR, 0);
int reftop  = max(curtop - SR, 0);
int cand_x = refleft + threadIdx.x;
int cand_y = reftop  + threadIdx.y;
int sad = 0;
__shared__ int3 sads[256];

for(int j = 0; j < MBSIZE; ++j) {
  for(int i = 0; i < MBSIZE; ++i){
    sad = __sad(ref[INDEX(cand_x+i, cand_y + j)],
                cur[INDEX(curleft+i,curtop + j)], sad);
  }
}
sads[tid] = make_int3(threadIdx.x, threadIdx.y, sad);
reduce_costs(sads);  // Parallel Reduction
if(tid == 0) {
  const int mvindex = (blockIdx.x + blockIdx.y * WIDTH/MBSIZE) * 2;
  mv[mvindex]   = sads[0].x - 8;
  mv[mvindex+1] = sads[0].y - 8;
}
```

Thread Blocks = 40x30
Threads = 16x16

CPU = 94.15 msec
GPU#1 = 111.21 msec (0.84x)



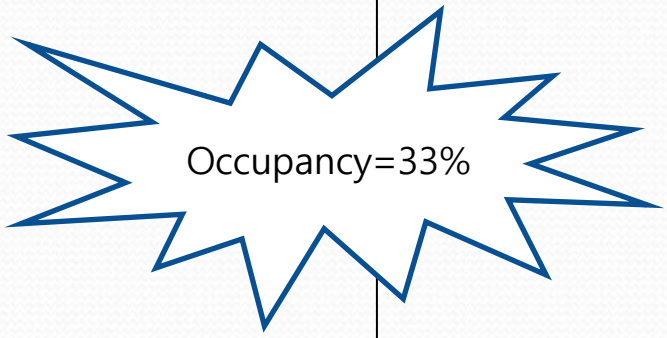
25,141,248

GPU Impl. #2 (Shared Memory)

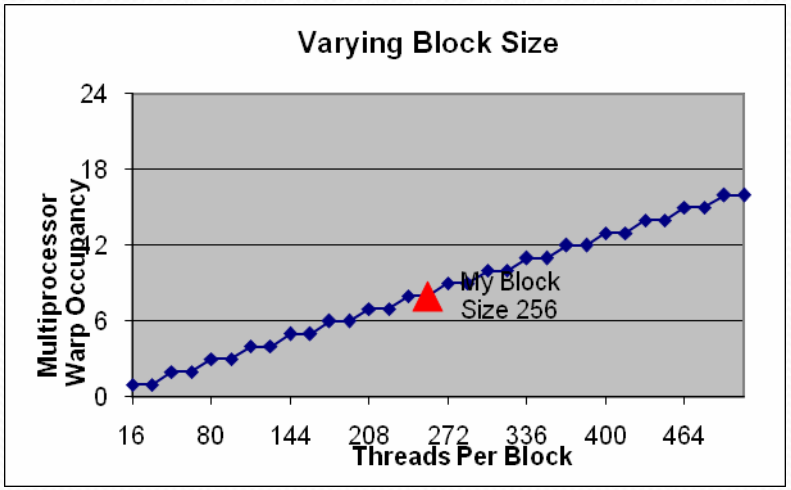
```
int tx = threadIdx.x;
int ty = threadIdx.y;
int tid = tx + ty * blockDim.x;
int curleft = blockIdx.x * MBSIZE;
int curtop  = blockIdx.y * MBSIZE;
int refleft = curleft - SR;
int reftop  = curtop - SR;
__shared__ int refcache[32][32];
__shared__ int curcache[16][16];
__shared__ int3 sads[256];

curcache[ty][tx] = cur[INDEX(curleft+tx,curtop+ty)];  // 현재 Block을 Shared로 Copy
for(int j = 0; j < 2; ++j)
  for(int i = 0; i < 2; ++i)
    refcache[ty*2+j][tx*2+i] = ref[INDEX(refleft+tx*2 +i, reftop+ty*2+j)];
__syncthreads();

int sad = 0;
for(int j = 0; j < MBSIZE; ++j) {
  for(int i = 0; i < MBSIZE; ++i) {
    sad = __sad(refcache[ty+j][tx+i], curcache[j][i], sad);
  }
}
sads[tid] = make_int3(threadIdx.x, threadIdx.y, sad);
reduce_costs(sads);
// 이하 생략
```

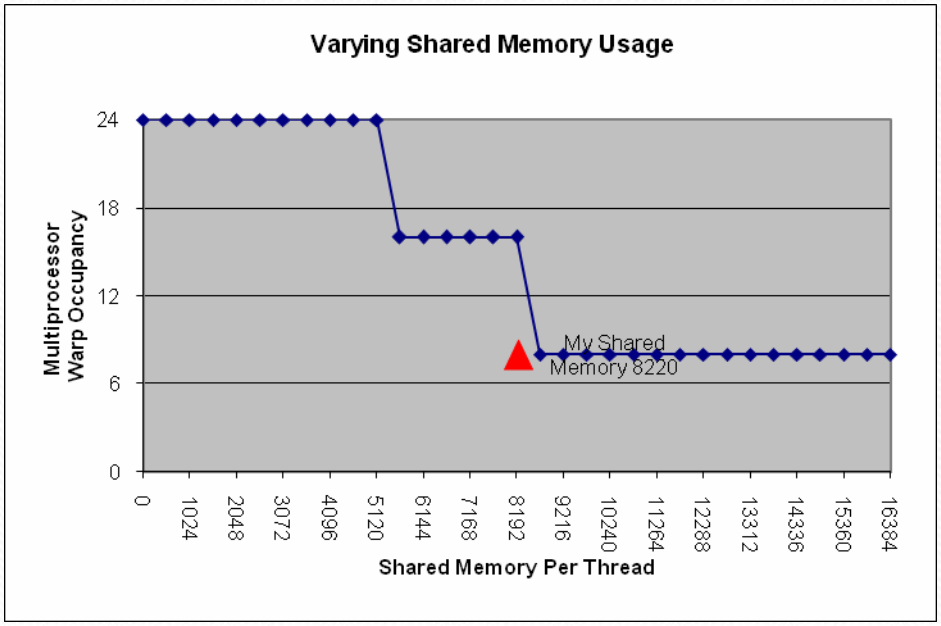
계산이 많은 블록을 Shared로 옮겨서 계산

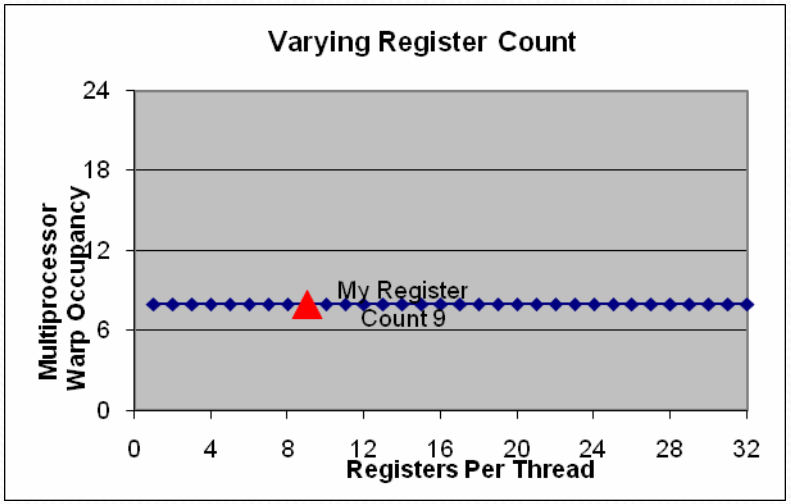
CPU = 94.15 msec
GPU#1 = 111.21 msec (0.84x)
GPU#2 = 3.27 msec (28.8x)



Check Occupancy







Threads / Block = 256
Registers / Thread = 9
Shared Memory / Block = 8220

Shared Memory를 조금만 줄이면
Occupancy 상승 !

GPU Impl. #3 (Raise Occupancy)

```
//
__shared__ int refcache[31][31];  // Search Range 1 감소
//
```

CPU = 94.15 msec
GPU#1 = 111.21 msec (0.84x)
GPU#2 = 3.27 msec (28.8x)
GPU#3 = 2.88 msec (32.7x)

Threads / Block = 256
Registers / Thread = 9
Shared Memory / Block = 7986

- CUDA Coding때 항상 염두에 둘 사항
  - Use shared memory
  - Evade global memory incoherent
  - Reduce register/shared memory size
  - Shared memory bank conflict
- 이론만 믿지 말고 이것 저것 해볼 것!

voceweb

More Expensive Cards

Model	GPU#1	GPU#2	GPU#3
8800GTS	111.21	3.27	2.88
8800GT	107.49 (3%)	2.23 (47%)	2.00 (44%)
Tesla C870	89.33 (24%)	2.02 (61%)	1.78 (62%)

Video Encoding에서는 0.1ms도 소중하다.

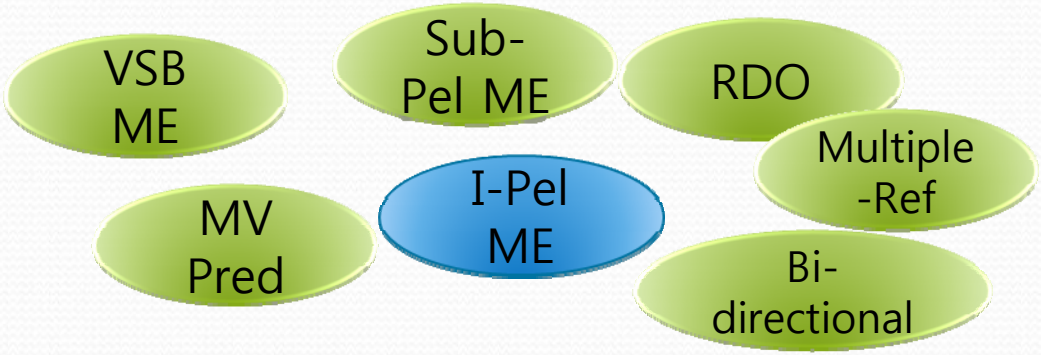
Model	SP	Bandwidth	Clock(C/S/M) MHz
8800GTS	96	64 GB/sec	500/1200/800
8800GT	112	57.6 GB/sec	600/1500/900
Tesla C870	128	76.8 GB/sec	600/1350/800

Issues on Parallel Video Encoding

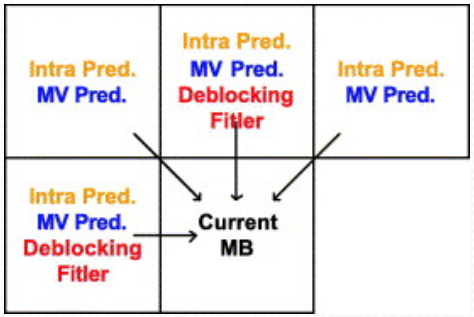
For H.264

Face the Real World : H.264

- Motion estimation is not simple!

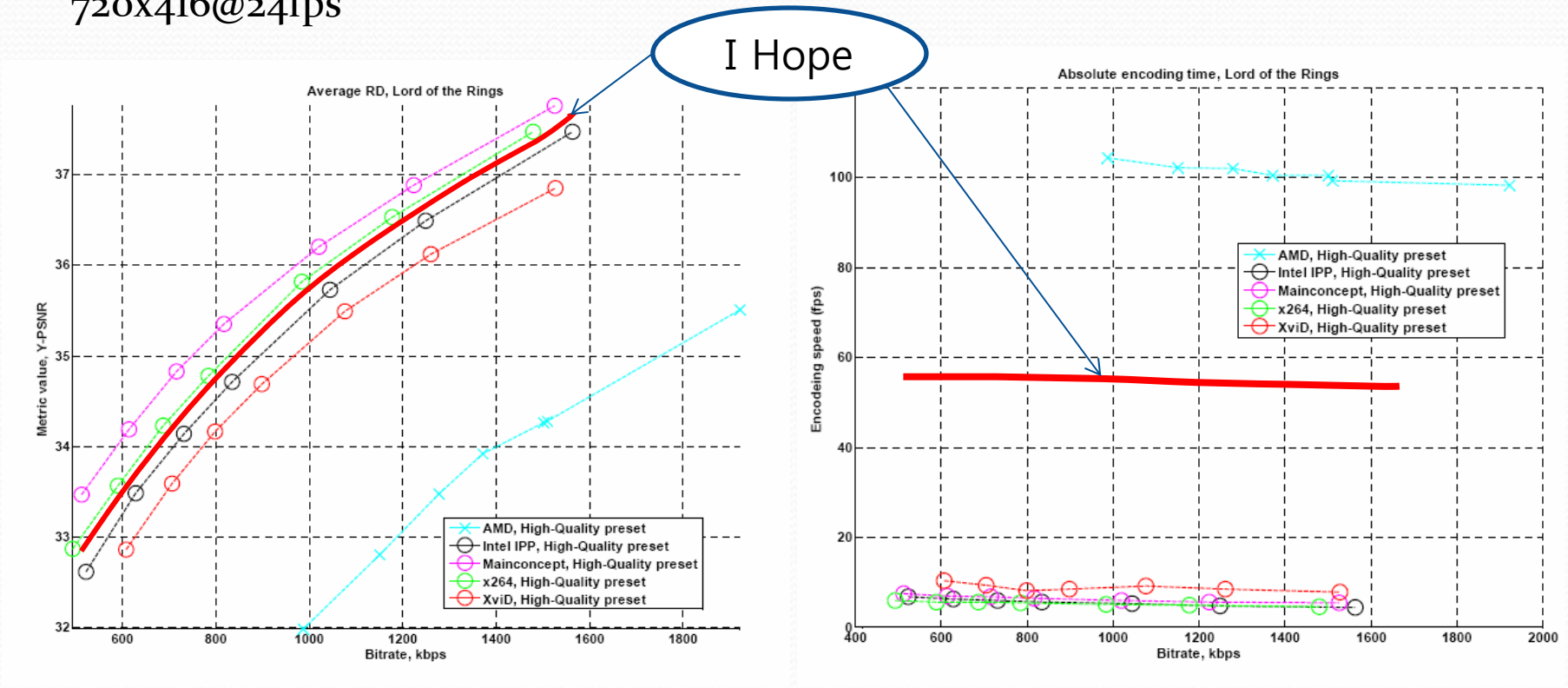


- H.264 encoding can't be parallel
  - Current MB depends on previous coded MB.
  - Algorithm & Structure modification ,Heuristics required!!!



H.264 Codec Comparison

720x416@24fps



- Algorithm 변경으로 인한 Quality Loss를 최소화하고 Encoding 속도를 극대화!!!

Video MPEG-4 AVC/H.264 Codec Comparison by MSU, Dec 2007

Some Intermediate Result

- Motion Estimation by Full Search
  - 36x faster than CPU
- Motion Estimation by UMH Fast Search
  - 6x faster than CPU
- Half-pel, Quarter-pel Building
  - 15x faster than CPU
- Overall Performance for High-Quality preset
  - 8x faster than JM reference
  - 0.2 dB maximum quality degradation

On GeForce 8800GT

Q&A