SIEMENS

NVIDIA GPU Technology Conference
March 18, 2015
San José, California

Avoiding Shared Memory Bank Conflicts in Rate Conversion Filtering

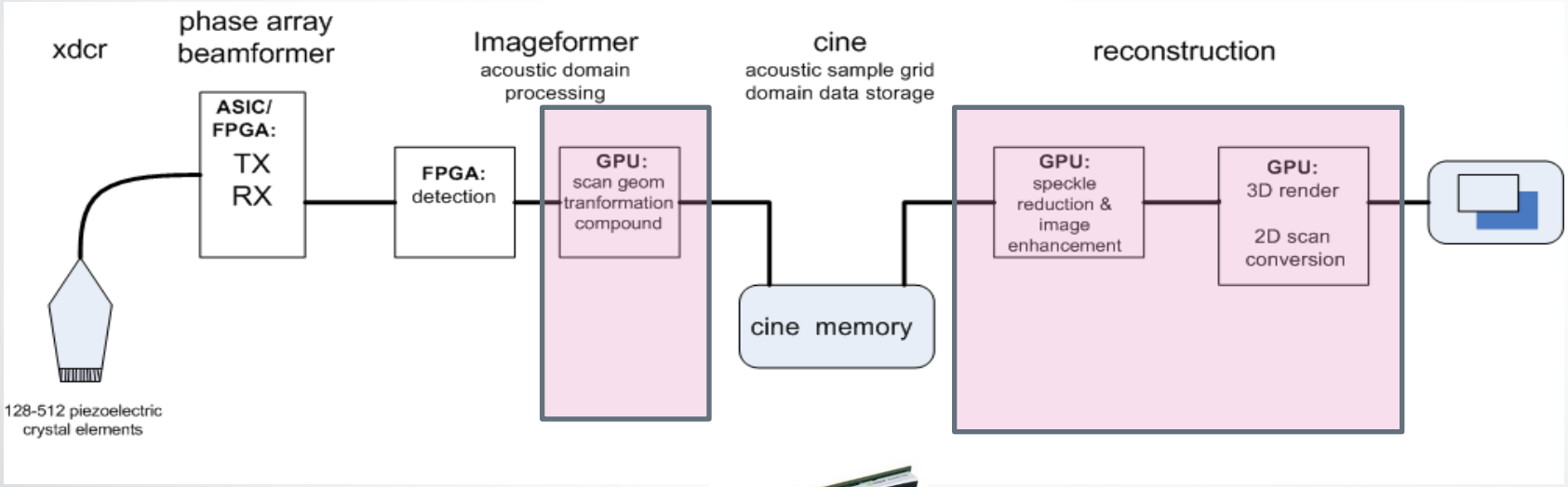
Mrugesh Gajjar
Technical Expert
Siemens Corporate Technology, India

Ismayil Güracar
Senior Key Expert
Siemens Medical Solutions, USA Inc.
Ultrasound Business Unit

GTC2015: S5282 Wednesday 10:00 am

SIEMENS

ACUSON SC2000™ Ultrasound System
Signal Processing Pipeline



SIEMENS

Image downsampling

Image downsampling is about resizing an image to smaller size in one or more dimension.

Referred by name ‘rate conversion’ in applications where data is of streaming nature such as audio.

Naïve downsampling by integer factor M: Select every M’th sample and drop the rest. It can result in aliasing.

SIEMENS

Rate conversion filtering

To remove aliasing: First low pass filter the data and then skip samples.

Filtering ranges for adjacent output pixels overlap.

Use shared memory to improve performance.

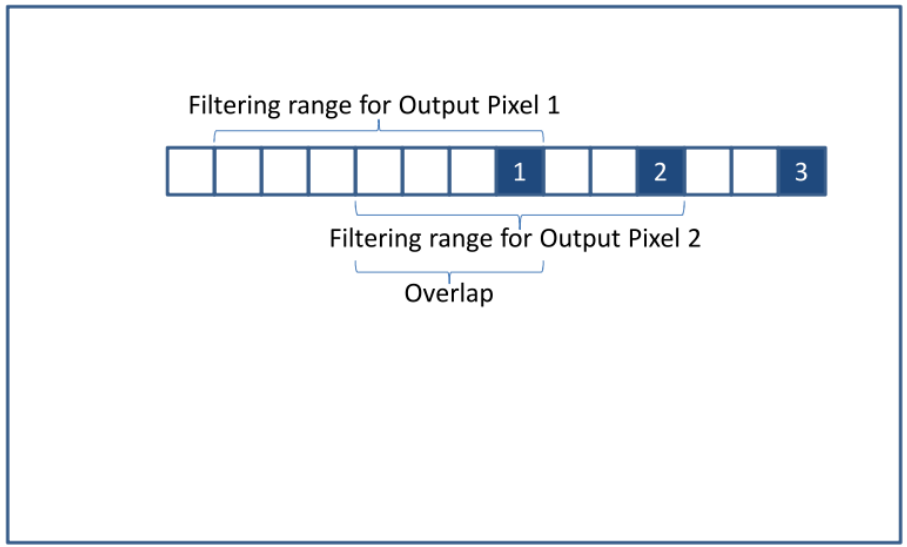
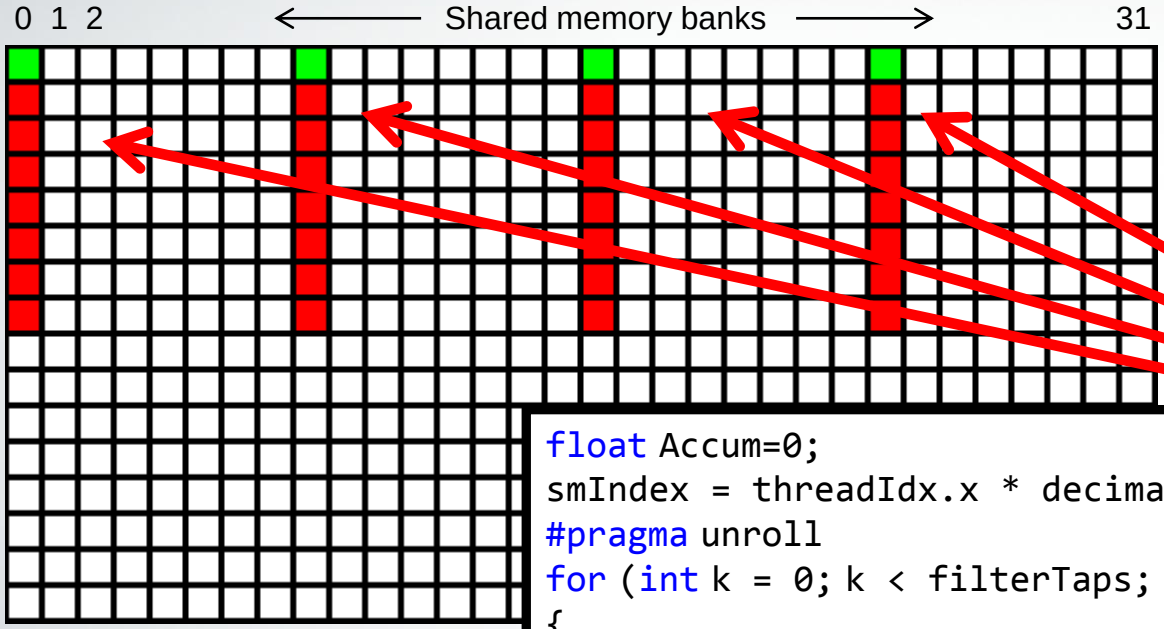


Illustration of 1/3 downsampling in X direction with Filter taps = 7

SIEMENS

Rate conversion filtering loop

Access pattern within a warp



No. of banks=32
decimationFactor=8

k=0

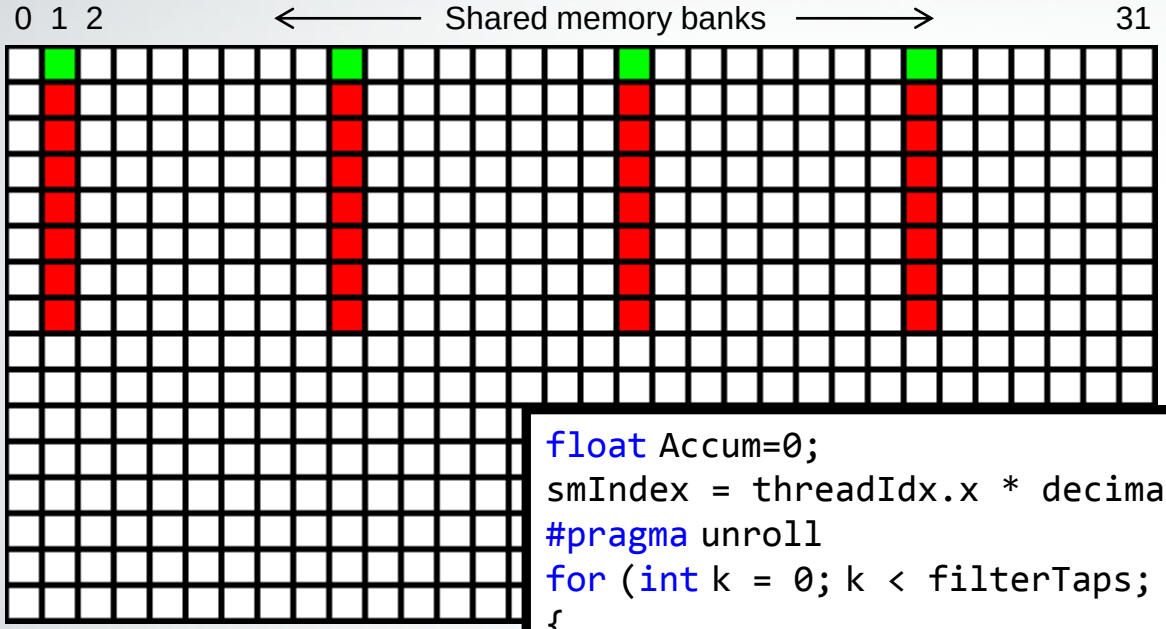
Shared Memory Bank conflicts

```
float Accum=0;
smIndex = threadIdx.x * decimationFactor;
#pragma unroll
for (int k = 0; k < filterTaps; k++)
{
  float C = d_coeff[k];
  Accum = Accum + C * sharedMem[k + smIndex];
}
```

SIEMENS

Rate conversion filtering loop

Access pattern within a warp



No. of banks=32
decimationFactor=8

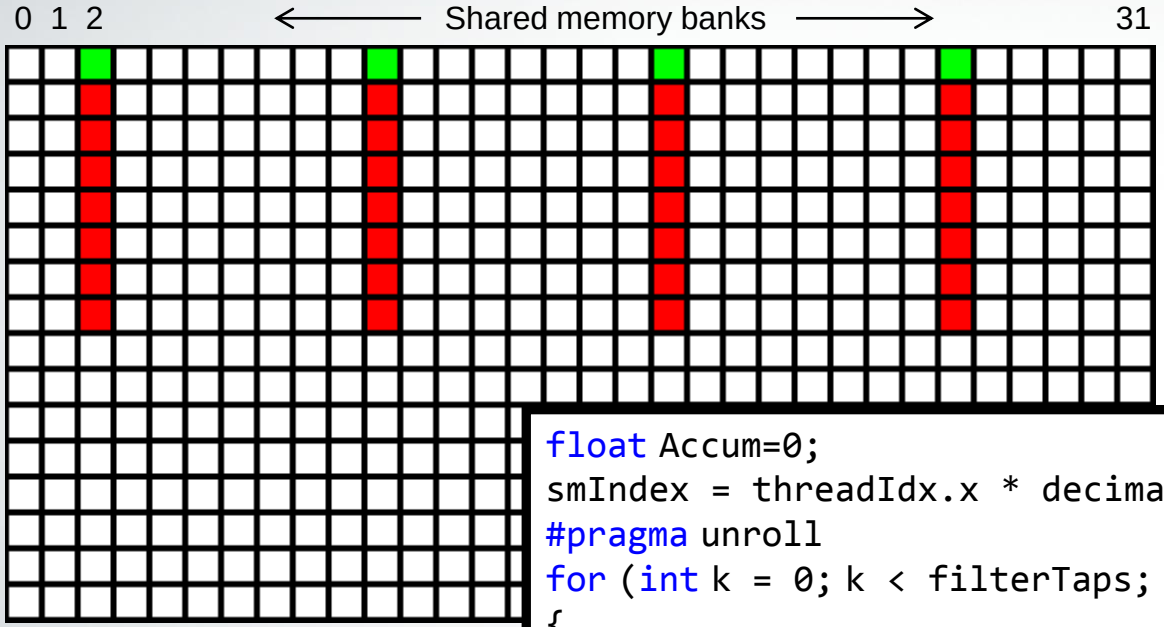
k=1

```
float Accum=0;
smIndex = threadIdx.x * decimationFactor;
#pragma unroll
for (int k = 0; k < filterTaps; k++)
{
  float C = d_coeff[k];
  Accum = Accum + C * sharedMem[k + smIndex];
}
```

SIEMENS

Rate conversion filtering loop

Access pattern within a warp



No. of banks=32
decimationFactor=8

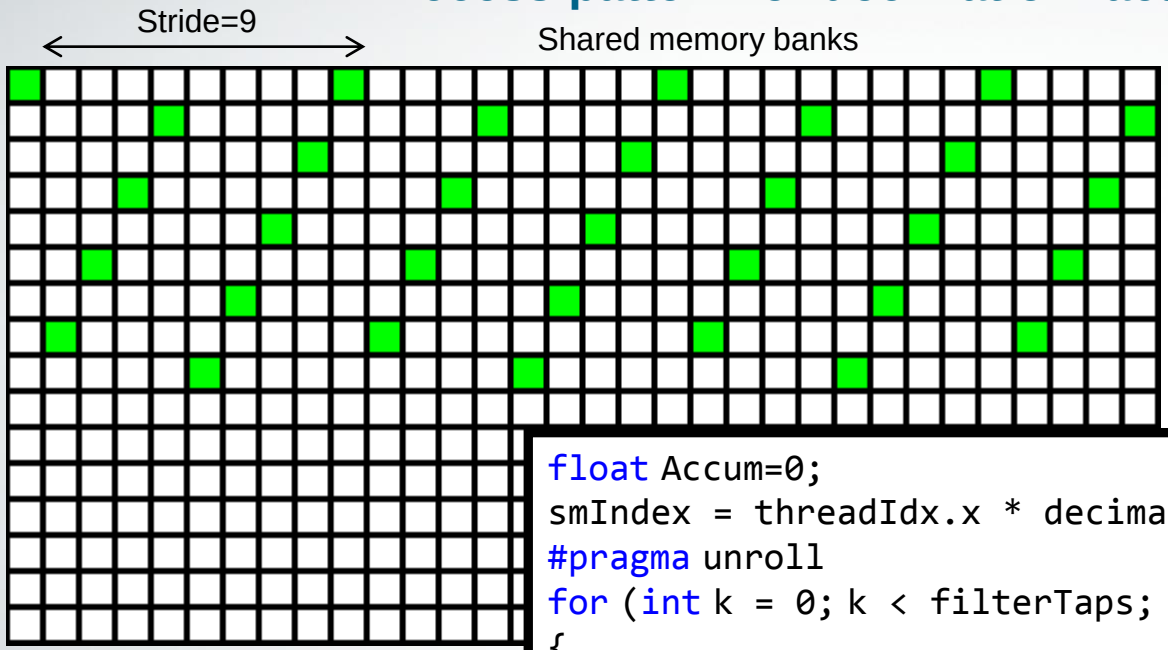
k=2

```
float Accum=0;
smIndex = threadIdx.x * decimationFactor;
#pragma unroll
for (int k = 0; k < filterTaps; k++)
{
  float C = d_coeff[k];
  Accum = Accum + C * sharedMem[k + smIndex];
}
```

SIEMENS

Rate conversion filtering loop

Access pattern for decimation factor 9



No. of banks=32
decimationFactor=9

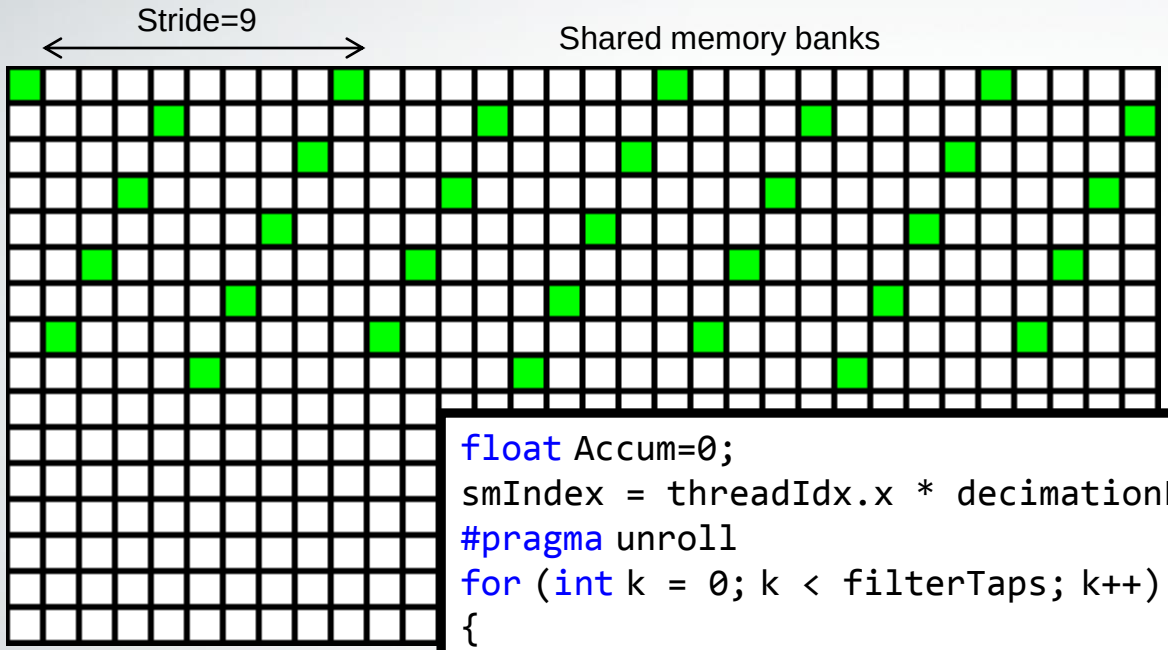
k=0

```
float Accum=0;
smIndex = threadIdx.x * decimationFactor;
#pragma unroll
for (int k = 0; k < filterTaps; k++)
{
  float C = d_coeff[k];
  Accum = Accum + C * sharedMem[k + smIndex];
}
```

SIEMENS

A solution

Dot product can be performed in any order



filterTaps=32
No. of banks=32
decimationFactor=8

k=0

```
float Accum=0;
smIndex = threadIdx.x * decimationFactor;
#pragma unroll
for (int k = 0; k < filterTaps; k++)
{
  int strided_k = (k + threadIdx.x) % filterTaps;
  float C = d_coeff[strided_k];
  Accum = Accum + C * sharedMem[strided_k + smIndex];
}
```

Thread i starts summing with index i in the dot product (convolution sum)

SIEMENS

Analysis

There will be bank conflicts if ***No. of banks (NB) and Downsampling factor (DF) shares a factor***.

Shared memory BW utilization reduces by that factor.

If NB and DF are co-prime there will not be bank conflicts.

Solution on previous slide: works as if we made DF=9, thus becoming co-prime with NB=32

SIEMENS

Analysis (contd.)

When NB is a power of 2, all even DF will result in bank conflicts.

Any odd number is co-prime to a power of 2. So, this solution will work.

```
float Accum=0;
smIndex = threadIdx.x * decimationFactor;
#pragma unroll
for (int k = 0; k < filterTaps; k++)
{
  int strided_k = (k + threadIdx.x) % filterTaps;
  float C = d_coeff[strided_k];
  Accum = Accum + C * sharedMem[strided_k + smIndex];
}
```

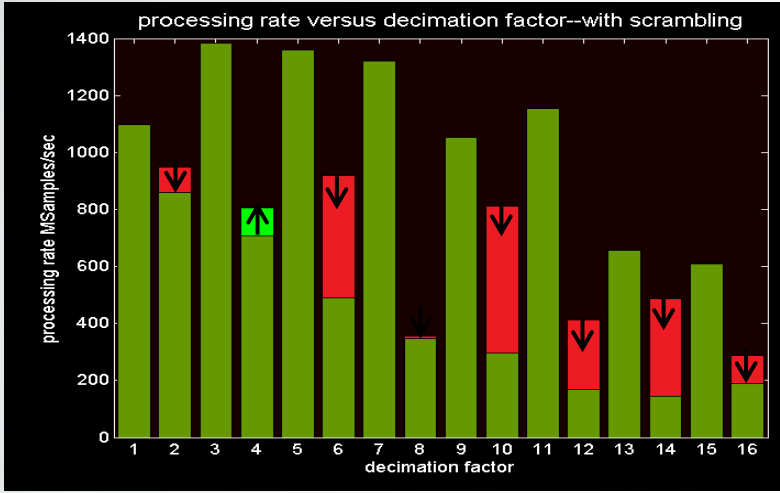
Two issues can hurt performance (especially for NVIDIA GPUs):

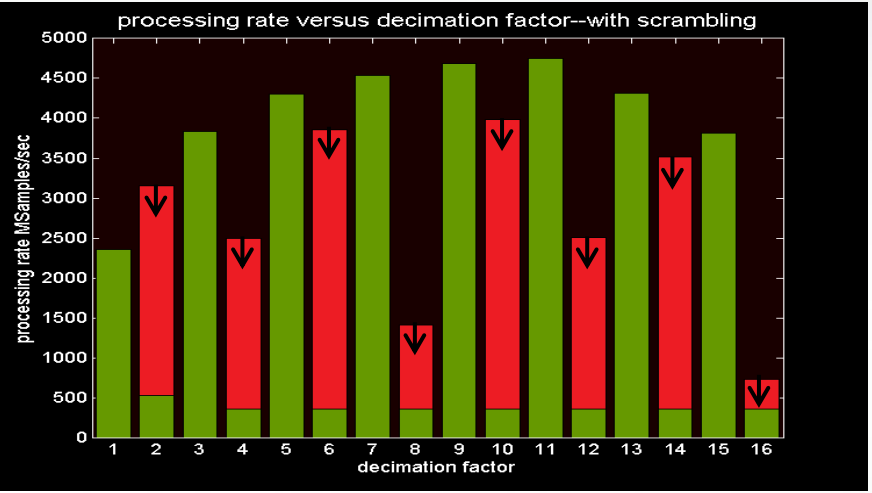
% operator in loop and

Slower access of coefficients due to divergent access in constant memory

SIEMENS

Performance result on Quadro 4000 (Fermi)
Basic striding using % operator





Input size: 4096 x 256, float4 samples

Input size: 4194304 x 1, float samples

The performance actually drops for almost all decimation factors.

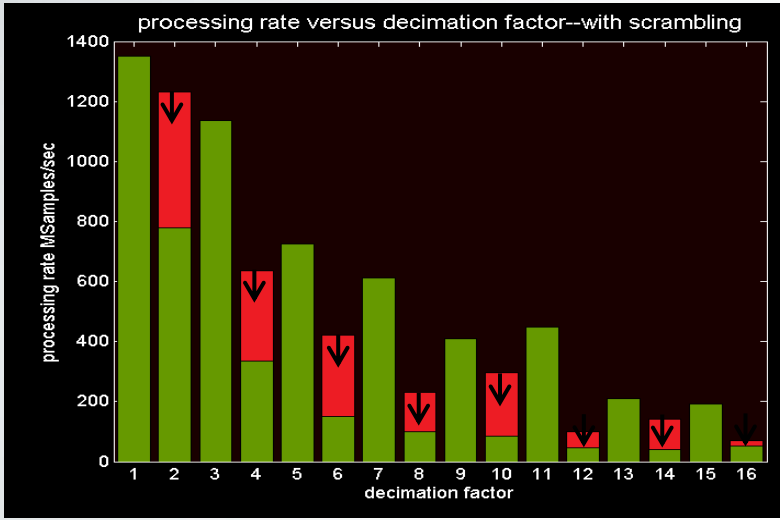
Reasons:

1) Costly % operator computation

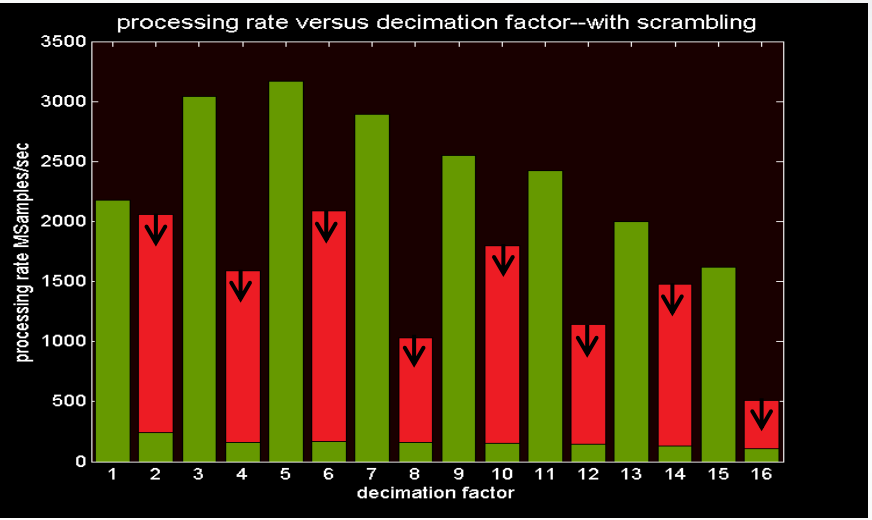
2) Threads read different coefficient locations, not allowing efficient broadcast.

SIEMENS

Performance result on Quadro K2000 (Kepler)
Basic striding using % operator



Input size: 4096 x 256, float4 samples



Input size: 4194304 x 1, float samples

Similar behavior.

However, slowdowns are much lower in case of float4 samples, as the cost of % operator is diluted by more compute work per sample.

SIEMENS

Can we do better?

Permutation scrambling

Load input from shared memory in scrambled manner using a different permutation per group of threads

Combine the co-efficients and shared memory inputs according to the permutation

```
iweight = (threadIdx.x & 0x18) >> 2;
fweight0 = 1.0 * (iweight == 0);
fweight1 = 1.0 * (iweight == 2);
fweight2 = 1.0 * (iweight == 4);
fweight3 = 1.0 * (iweight == 6);
Accum = 0;
for (k = 0; k < filterTaps; k += 8)
{
        C0 = C[k];
        C1 = C[k+1];
        C2 = C[k+2];
        C3 = C[k+3];
        C4 = C[k+4];
        C5 = C[k+5];
        C6 = C[k+6];
        C7 = C[k+7];
        n = k + threadIdx.x * decimationFactor;
        X0 = sharedMem[n + (0 ^ iweight)];
        X1 = sharedMem[n + (1 ^ iweight)];
        X2 = sharedMem[n + (2 ^ iweight)];
        X3 = sharedMem[n + (3 ^ iweight)];
        X4 = sharedMem[n + (4 ^ iweight)];
        X5 = sharedMem[n + (5 ^ iweight)];
        X6 = sharedMem[n + (6 ^ iweight)];
        X7 = sharedMem[n + (7 ^ iweight)];
        Accum = Accum +
        (C0*X0+C1*X1+C2*X2+C3*X3 + C4*X4+C5*X5+C6*X6+C7*X7) * fweight0 +
        (C0*X2+C1*X3+C2*X0+C3*X1 + C4*X6+C5*X7+C6*X4+C7*X5) * fweight1 +
        (C0*X4+C1*X5+C2*X6+C3*X7 + C4*X0+C5*X1+C6*X2+C7*X3) * fweight2 +
        (C0*X6+C1*X7+C2*X4+C3*X5 + C4*X2+C5*X3+C6*X0+C7*X1) * fweight3 ;
}
```

Divide a warp into 4 groups of threads

Tile the convolution loop

Within a tile: **Load all coefficients sequentially** in registers beforehand

SIEMENS

Permutation scrambling

Shared memory access locations within a warp, within a tile

Before

Time steps

Threads 0 .. 7	0	1	2	3	4	5	6	7
Threads 8 .. 15	0	1	2	3	4	5	6	7
Threads 16 .. 23	0	1	2	3	4	5	6	7
Threads 24 .. 31	0	1	2	3	4	5	6	7

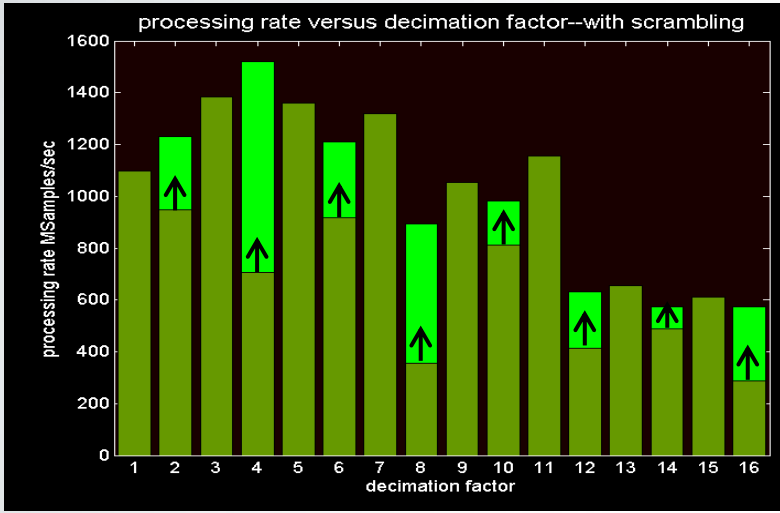
After

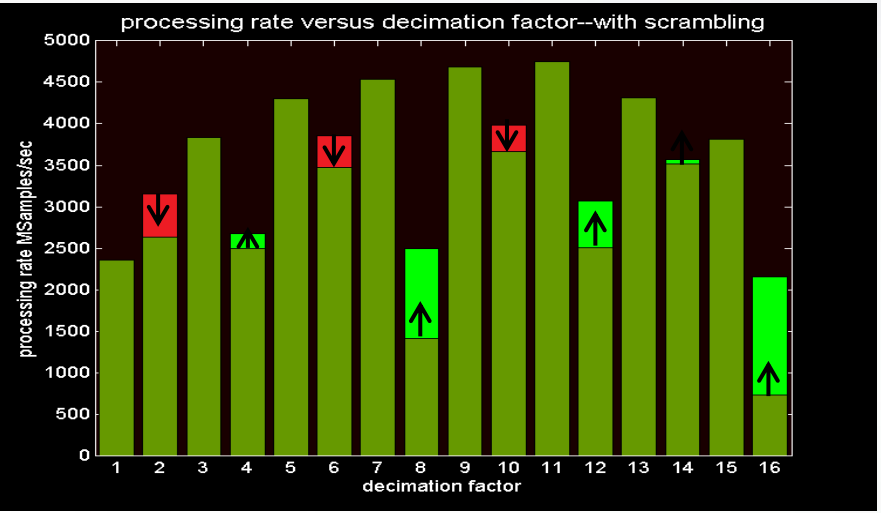
Time steps

0	1	2	3	4	5	6	7
2	3	0	1	6	7	4	5
4	5	6	7	0	1	2	3
6	7	4	5	2	3	0	1

SIEMENS

Performance result on Quadro 4000 (Fermi)
Permutation scrambling





Input size: 4096 x 256, float4 samples

Performance improves for all even downsampling factors

Speedups: 1.18x to 2.51x

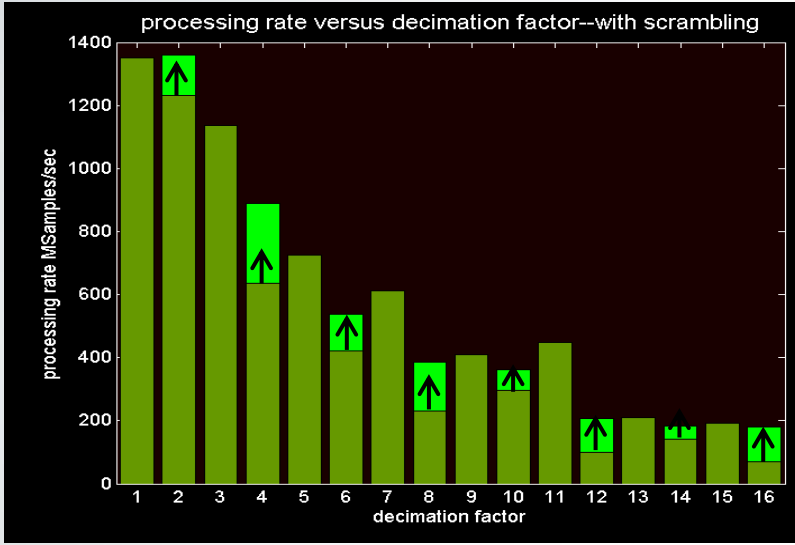
Input size: 4194304 x 1, float samples

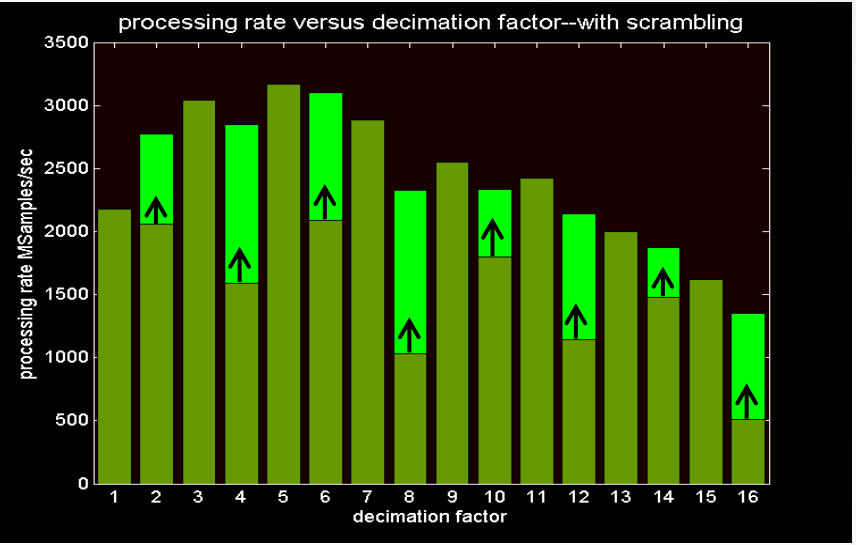
Performance only improves for higher amount of bank conflicts. E.g., DF=4,8,12,16

Speedups: 0.84x to 2.95x

SIEMENS

Performance result on Quadro K2000 (Kepler)
Permutation scrambling





Input size: 4096 x 256, float4 samples

Performance improves for all even downsampling factors

Speedups: 1.10x to 2.63x

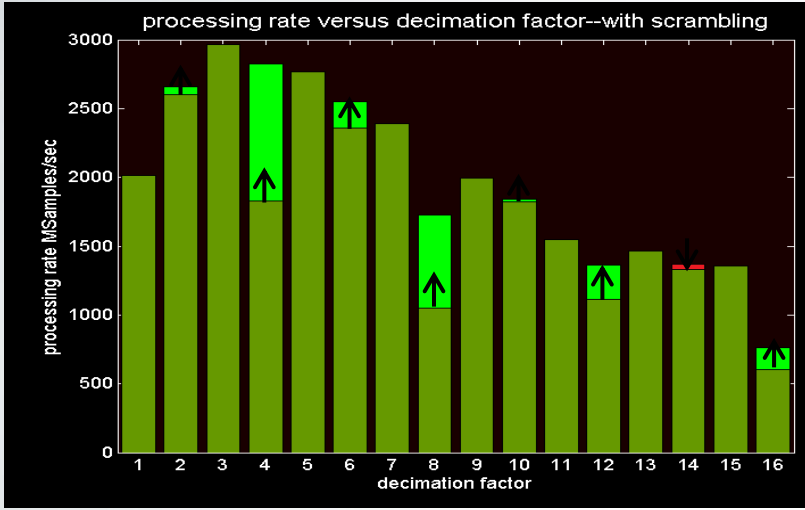
Input size: 4194304 x 1, float samples

Performance improves for all even downsampling factors

Speedups: 1.27x to 2.66x

SIEMENS

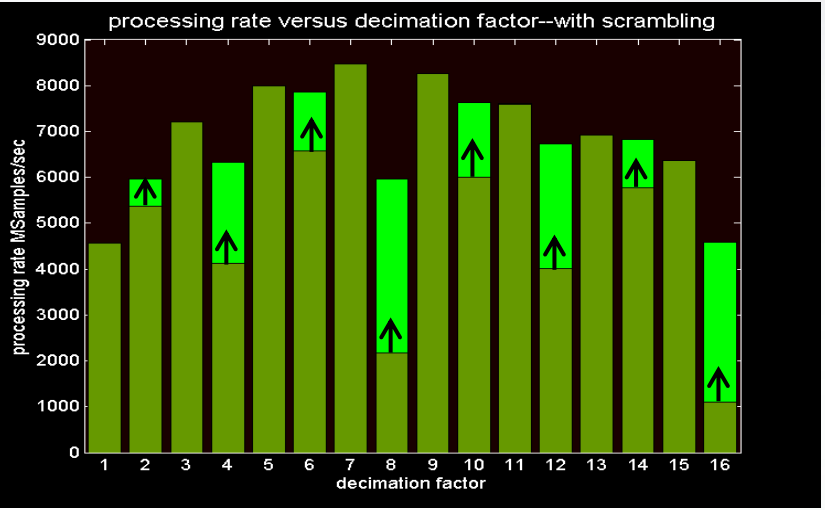
Performance result on Quadro K2200 (Maxwell)
Permutation scrambling



Input size: 4096 x 256, float4 samples

Performance improves for all even downsampling factors except 14

Speedups: 0.97x to 1.65x



Input size: 4194304 x 1, float samples

Performance improves for all even downsampling factors

Speedups: 1.11x to 4.15x

SIEMENS

Correlation between Speedups and DF with Permutation scrambling (4096 x 256 x float4)

Consistently higher speedups for DF=4,8,12,16 because bandwidth loss was 4x or more

Downsampling factor (DF)	Theoretical shared memory BW loss	Quadro 4000 (Fermi) Speedup	Quadro K2000 (Kepler) Speedup	Quadro K2200 (Maxwell) Speedup
2	2x	1.30x	1.10x	1.02x
4	**4x**	**2.15x**	**1.40x**	**1.54x**
6	2x	1.32x	1.28x	1.08x
8	**8x**	**2.51x**	**1.68x**	**1.65x**
10	2x	1.21x	1.22	1.01x
12	**4x**	**1.53x**	**2.10x**	**1.22x**
14	2x	1.18x	1.30x	0.97x
16	**16x**	**2.00x**	**2.63x**	**1.25x**

SIEMENS

Correlation between Speedups and DF with Permutation scrambling (4194304 x 1 x float)

Consistently higher speedups for DF=4,8,12,16 because bandwidth loss was 4x or more

Downsampling factor (DF)	Theoretical shared memory BW loss	Quadro 4000 (Fermi) Speedup	Quadro K2000 (Kepler) Speedup	Quadro K2200 (Maxwell) Speedup
2	2x	0.84x	1.35x	1.11x
4	**4x**	**1.07x**	**1.79x**	**1.53x**
6	2x	0.90x	1.48x	1.19x
8	**8x**	**1.77x**	**2.25x**	**2.73x**
10	2x	0.92x	1.30x	1.27x
12	**4x**	**1.22x**	**1.89x**	**1.68x**
14	2x	1.01x	1.27x	1.18x
16	**16x**	**2.95x**	**2.66x**	**4.15x**

SIEMENS

Summary & next steps

The ideas presented are quite general and applicable for other GPUs too.

Characterize performance on Intel, AMD GPUs

General solution for any number of banks

Explore fractional rate conversion, 2D filters, effects of ILP and data item size

SIEMENS

Thank you for your attention, Questions?

Mrugesh Gajjar
Siemens Medical Solutions, USA Inc.
Ultrasound Business Unit

685 E. Middlefield Road
Mountain View, CA 94043
mrugesh.gajjar@siemens.com