

# **Experiences Porting Real Time Signal Processing Pipeline CUDA Kernels from Kepler to Maxwell**

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

**GTC2015 S5223 Friday 9:30 am**

# Diagnostic Ultrasound Imaging Equipment

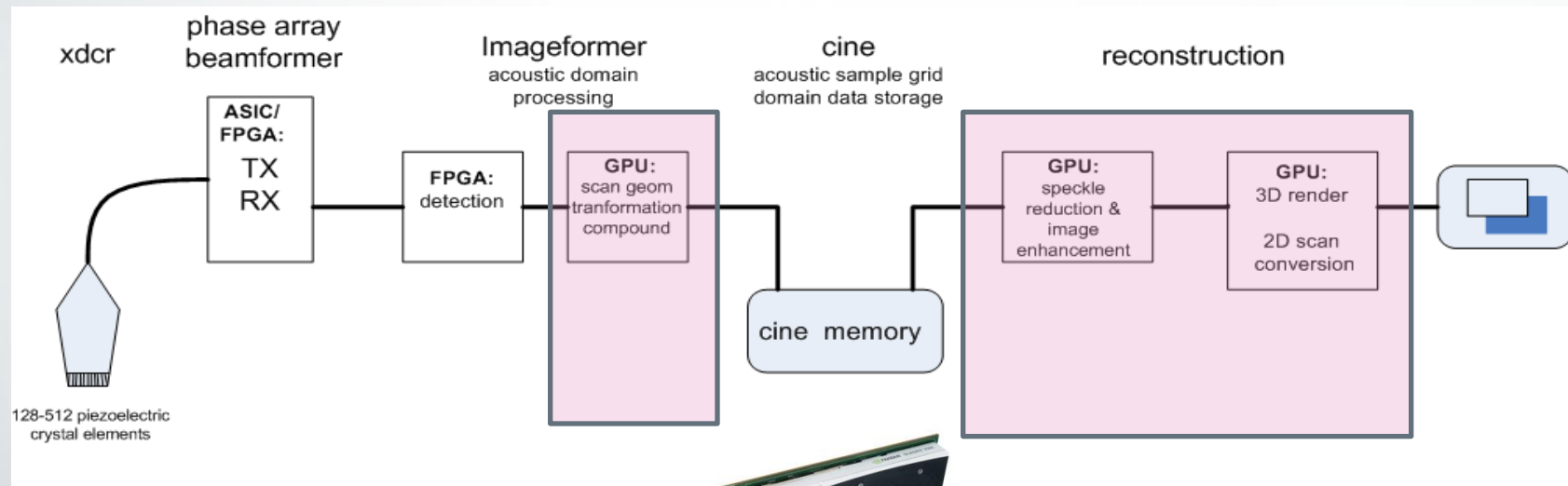


A machine for the acquisition of imaging information to affect diagnosis and treatment



# ACUSON SC2000™ Ultrasound System

## Signal Processing Pipeline



# Medical Instrument Programming and Hardware Environment:

- Long lifetime in the marketplace (> decade)
- Constant need to respond to change over the product lifetime
  - 2008 WinXP CUDA 2.3 GeForce 9800GT
  - 2011 WinXP CUDA 2.3 Quadro 2000
  - 2014 Win8 CUDA 5.5 Kepler K2000
  - 2015 Win8 CUDA 6.5 Maxwell K2200

## Changes needed to be ready for Maxwell:

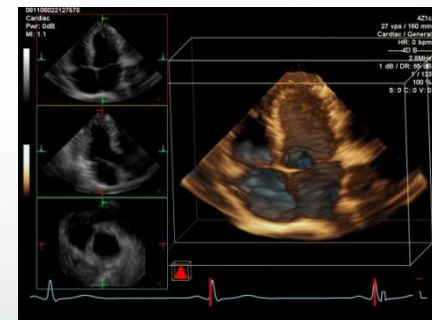
- Requires new display driver ( $\geq 340.52$ )
- Compute Capability 5.0  
Not object code compatible with earlier code
- If we try to run the old CC 3.0 compiled code the driver will JIT compile from PTX to generate CC 5.0 code for Maxwell  
There will be a big delay on startup!  
Better to recompile with CUDA 6.5 (or later) and generate CC 5.0 GPU object code

# A First Look at Performance

Just plug in a Maxwell K2200 and try it out

Relative processing rate -- selected imaging conditions

	Kepler K2000	Maxwell K2200
Application #1 2D Speckle Reduction	100%	<b>146%</b>
Application #2 2D Spatial Compound	100%	<b>113%</b>
Application #3 3D Speckle Reduction	100%	<b>130%</b>



# Instruction Level Parallelism Experiments: Kepler versus Maxwell

## ➤ Performance Improvement from Specifications

### Kepler K2000

2 SMX × 192 cores/SMX

954 MHz × 384 cores × 2 (FMA)  
= 733 Gflops



### Maxwell K2200

5 SMM × 128 cores/SMM

1124 MHz × 640 cores × 2 (FMA)  
= 1439 Gflops

## ➤ What is actually achievable?

## ➤ ILP experiments

To learn more about ILP, see

Volkov, “Better Performance at Lower Occupancy”

<http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf>

## Instruction Level Parallelism (ILP) Experiments

```
#define N_ITERATIONS 100
#define INTERNAL_ITERATIONS 100

__global__ void
ilp1_kernel(float *d_In, float *d_Out)
{
    float a = d_In[threadIdx.x];
    float b = d_In[threadIdx.x + 1];
    float c = d_In[threadIdx.x + 2];

    for (int x=0; x<INTERNAL_ITERATIONS; x++)
    {
        #pragma unroll
        for (int y=0; y<N_ITERATIONS; y++)
        {
            a = a*b + c;
        }
    }
    d_Out[ii]=a;
}
```

Inner loop assembly code fragment

```
...
FFMA R6, R6, R3, R2;
FFMA R6, R6, R3, R2;
FFMA R6, R6, R3, R2;
...
```

← Loops contain purely computation -- no I/O



## Instruction Level Parallelism (ILP) Experiments


Inner loop assembly code fragment

```
...
FFMA R8, R8, R5, R4;
FFMA R9, R9, R3, R2;
FFMA R8, R8, R5, R4;
FFMA R9, R9, R3, R2;
...
```

```
__global__ void
ilp2_kernel(float *d_In, float *d_Out)
{
    float a = d_In[threadIdx.x];
    float b = d_In[threadIdx.x + 1];
    float c = d_In[threadIdx.x + 2];
    float d = d_In[threadIdx.x + 3];
    float e = d_In[threadIdx.x + 4];
    float f = d_In[threadIdx.x + 5];

    for (int x=0; x<INTERNAL_ITERATIONS; x++)
    {
        #pragma unroll
        for (int y=0; y<N_ITERATIONS; y++)
        {
            a = a*b + c;
            d = d*e + f;
        }
    }
    d_Out[threadIdx.x]=a;
    d_Out[threadIdx.x]=d;
}
```

**a = a\*b + c;**  
**d = d\*e + f;**



No dependency between operations gives  
 The opportunity for instruction level parallelism  
 2-way ILP

# Instruction Level Parallelism (ILP) Experiments

```
__global__ void
ilp3_kernel(float *d_In, float *d_Out)
{
    ... initialize variables, setup loop

    #pragma unroll
    for (int y=0; y<N_ITERATIONS; y++)
    {
        a = a*b + c;
        d = d*e + f;
        g = g*h + i;

    }
    ... complete loop & output a,d and g
}
```

**a = a\*b + c;**

**d = d\*e + f;**

**g = g\*h + i;**

3-way ILP

```
ilp4_kernel(float *d_In, float *d_Out)
```

**a = a\*b + c;**

**d = d\*e + f;**

**g = g\*h + i;**

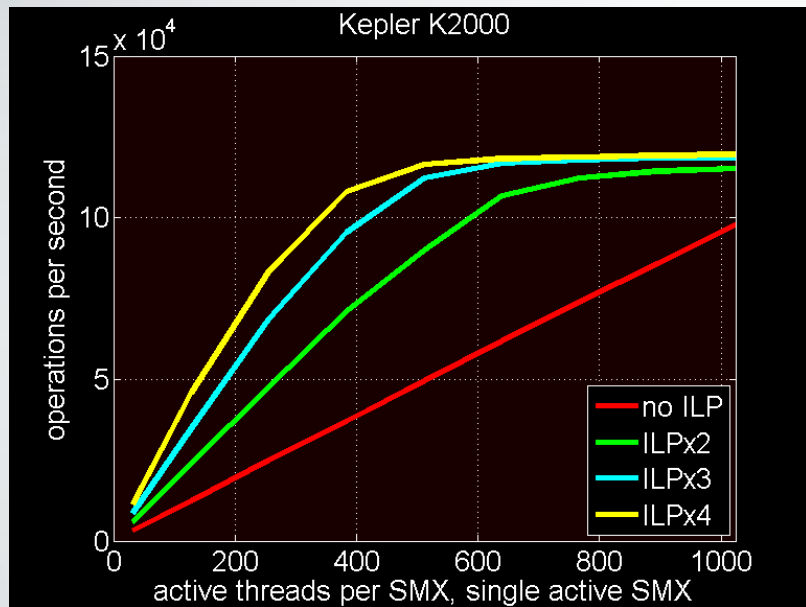
**j = j\*k + l;**

4-way ILP

# ILP Experiment Kernel Launch Arguments

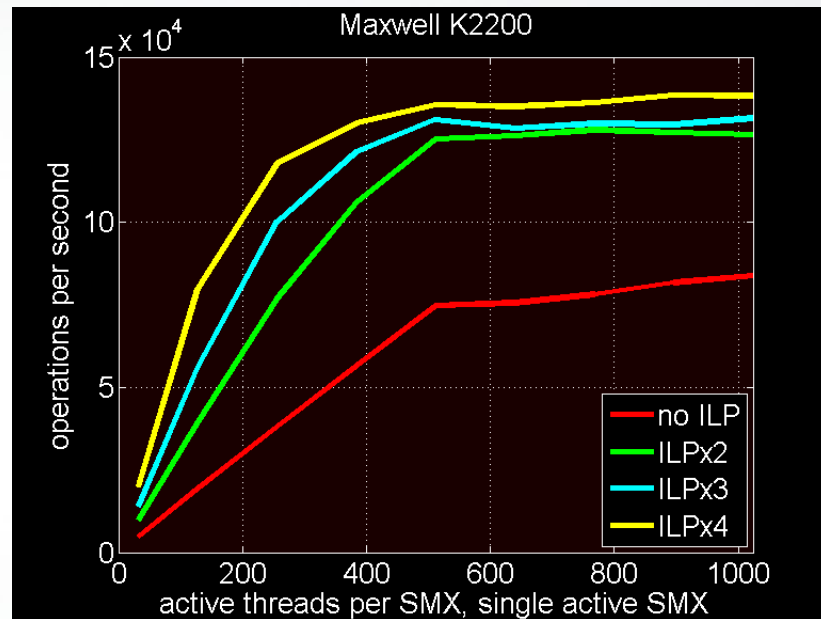
```
extern "C"  
void ilp1 (float *d_in, float *d_out, int threadCount)  
{  
    dim3 gridSz(1); // launch 1 thread block so only one SM will be active  
    dim3 blockSz(threadCount);  
    ilp1_kernel<<<gridSz,blockSz>>>(d_in, d_out);  
}
```

# ILP: How much is needed to achieve full capability?



## Kepler K2000

2 SMX  $\times 12 \times 10^4$  ops/sec



## Maxwell K2200

5 SMM  $\times 14 \times 10^4$  ops/sec

2.9x

## Conclusions on the Multiprocessor and ILP

- Overall Maxwell K2200 maximum instruction throughput is nominally about 2× Kepler K2000,
- Occupancy and ILP essential for providing enough work for threads in order to hide instruction execution latency

## Device Memory Bandwidth: Kepler versus Maxwell

	Quadro Kepler K2000	Quadro Maxwell K2200
Total Device Memory	2 GB	4 GB
Memory Clock	4.0 GHz	5.0 GHz
Memory Bus Width	128-bit	128-bit
Nominal Memory Bandwidth	64 GB/s	80 GB/s

# Kernel-based Device-to-Device Memory Copy Experiment

How does a purely I/O bound task scale with active threads?

Experiment: Varying amounts of bytes to copy (work) per thread to perform a large device to device memory copy

# 1 byte read / 1 byte write per thread

```
__global__ void
mem1_kernel(char *d_In, char *d_Out, int pitch)
{
    int ii = threadIdx.x + blockIdx.x * pitch;
    d_Out[ii]=d_In[ii];
}

extern "C"
void mem1(char *d_in, char *d_out, int bytesToCopy,
          int threadCount, int sharedMemPerThreadBlock)
{
    int blocks = bytesToCopy/threadCount;

    int pitch=threadCount;
    dim3 gridSz(blocks);
    dim3 blockSz(threadCount);
    mem1_kernel<<<gridSz,blockSz, sharedMemPerThreadBlock>>>(d_in, d_out,
                                                                pitch);
}
```



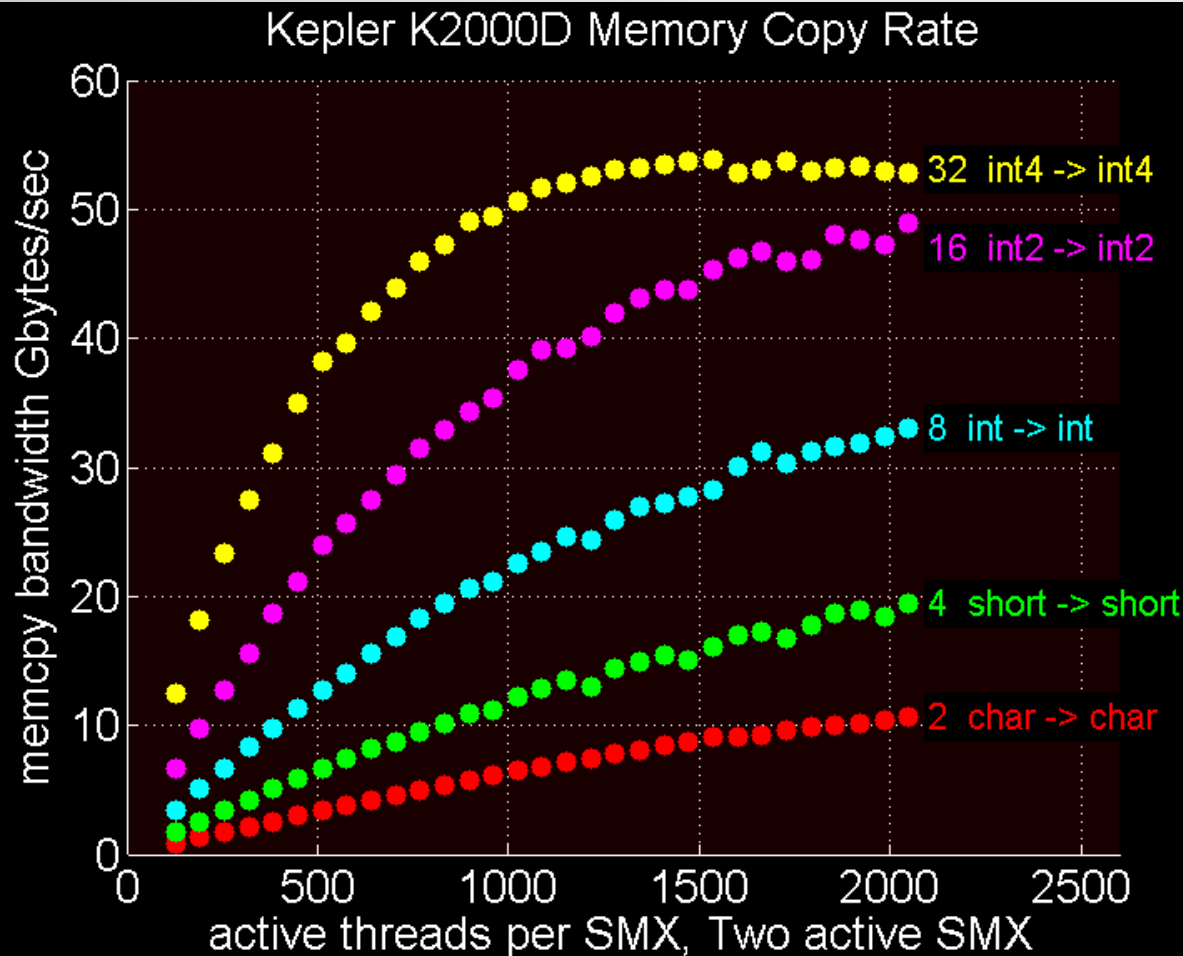
## 2 bytes read / 2 bytes write per thread

```
__global__ void  
mem2_kernel(short *d_In, short *d_Out, int pitch)  
{  
    int ii = threadIdx.x + blockIdx.x * pitch;  
    d_Out[ii]=d_In[ii];  
}
```

## Non-power of 2 R/W access    char3→char3

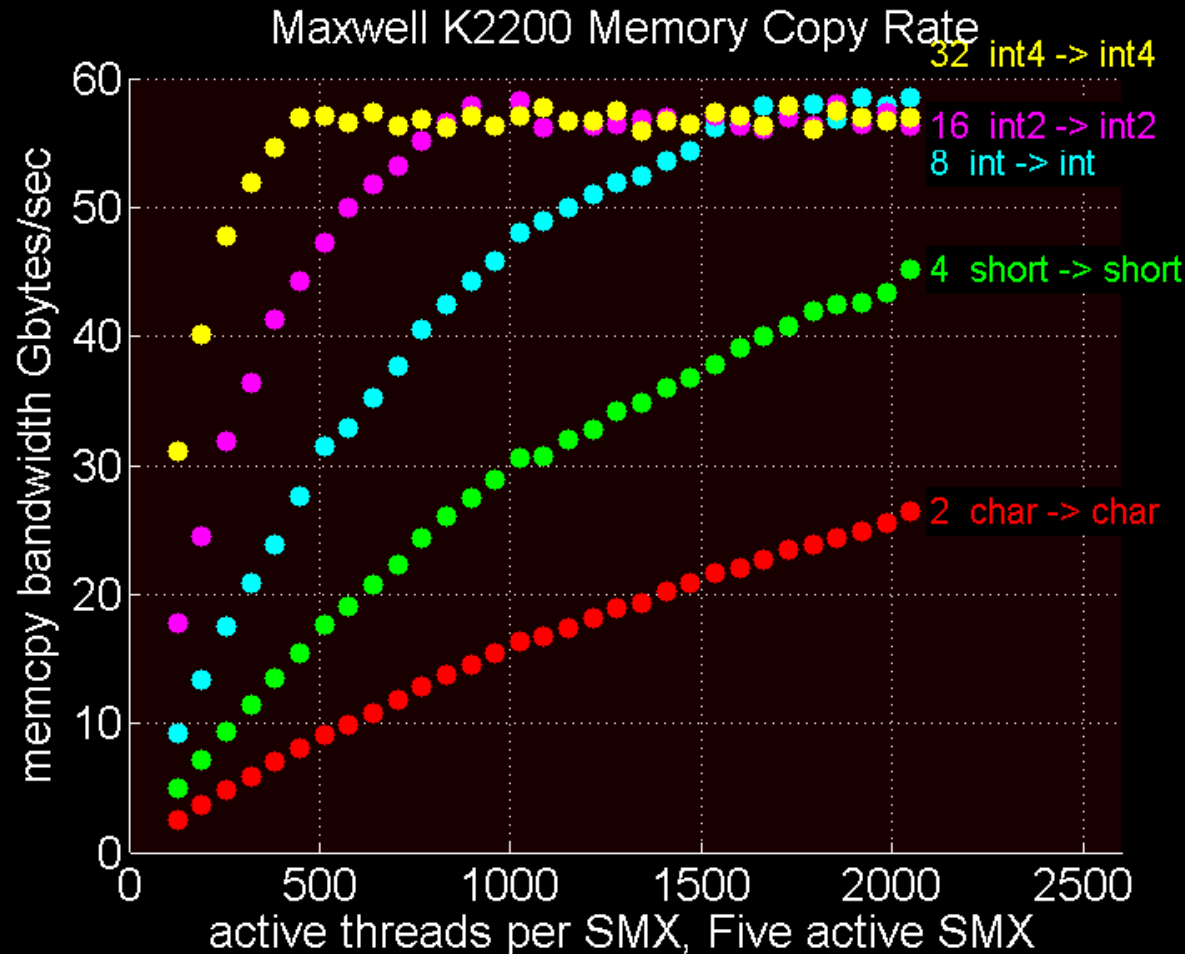
```
__global__ void  
mem3_kernel(char3 *d_In, char3*d_Out, int pitch)  
{  
    int ii = threadIdx.x + blockIdx.x * pitch;  
    d_Out[ii]=d_In[ii];  
}
```

## Kepler K2000



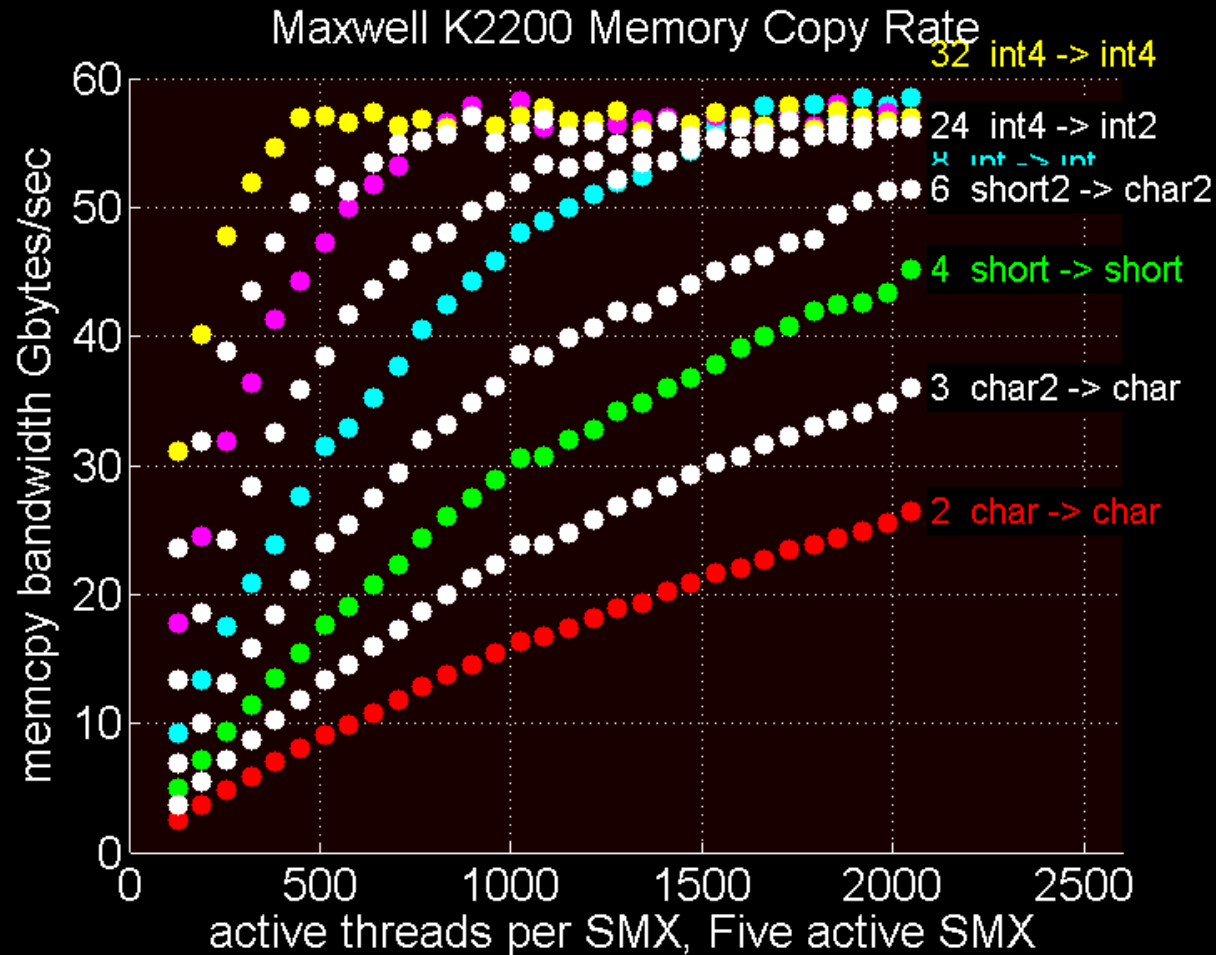
**Memcpy bandwidth  
Increases with more  
work (bytes moved)  
per thread until bus  
is saturated**

Saturation  
requires about 1k  
active threads per  
SMX and 32 read-  
write bytes per thread

**Maxwell K2200**

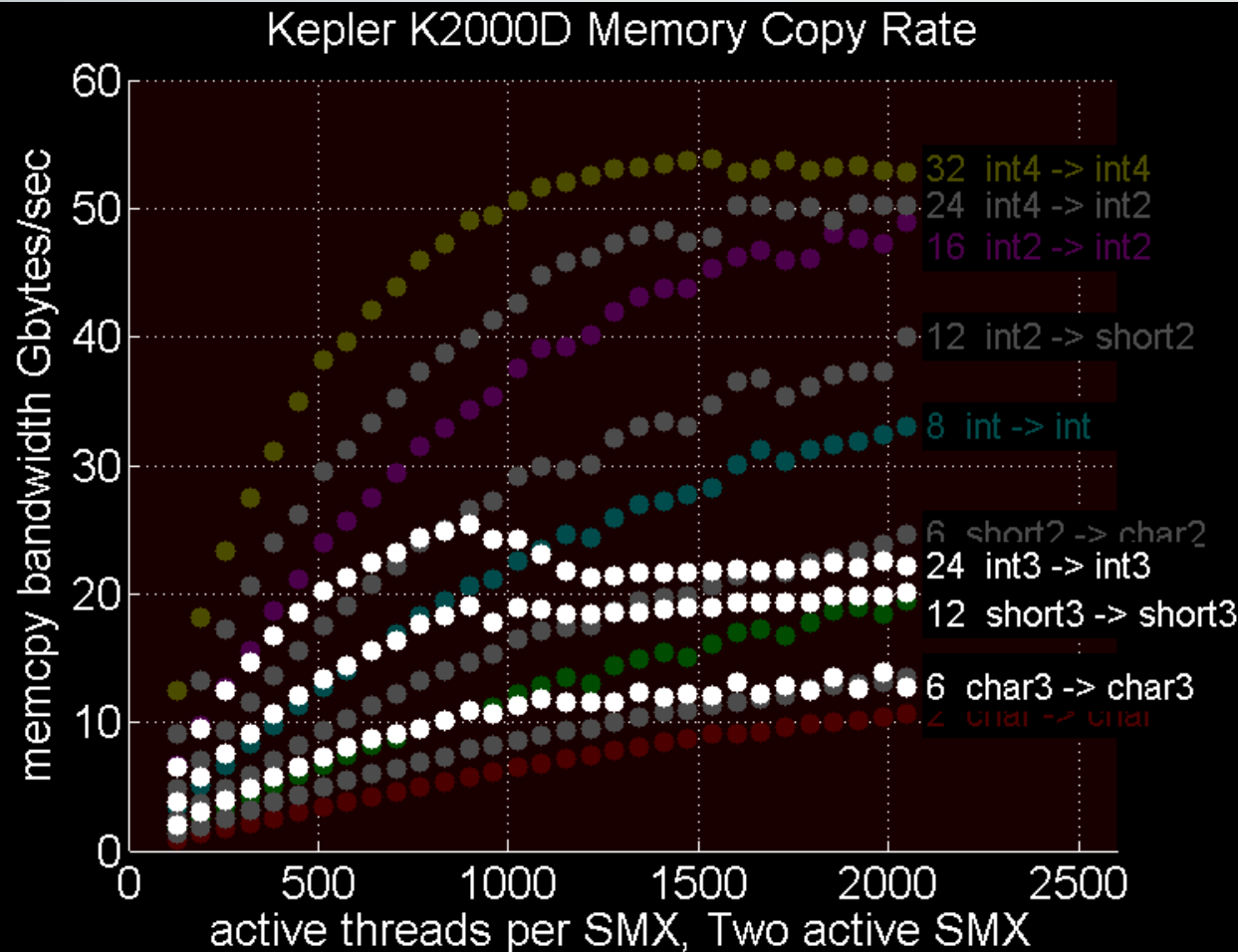
Saturation achieved  
With fewer active  
threads per SMX  
requires about 512  
active threads per  
SMX and 32 read-write  
bytes per thread

## Maxwell K2200



**Memcopy bandwidth  
Increases with more  
work (bytes moved)  
per thread**

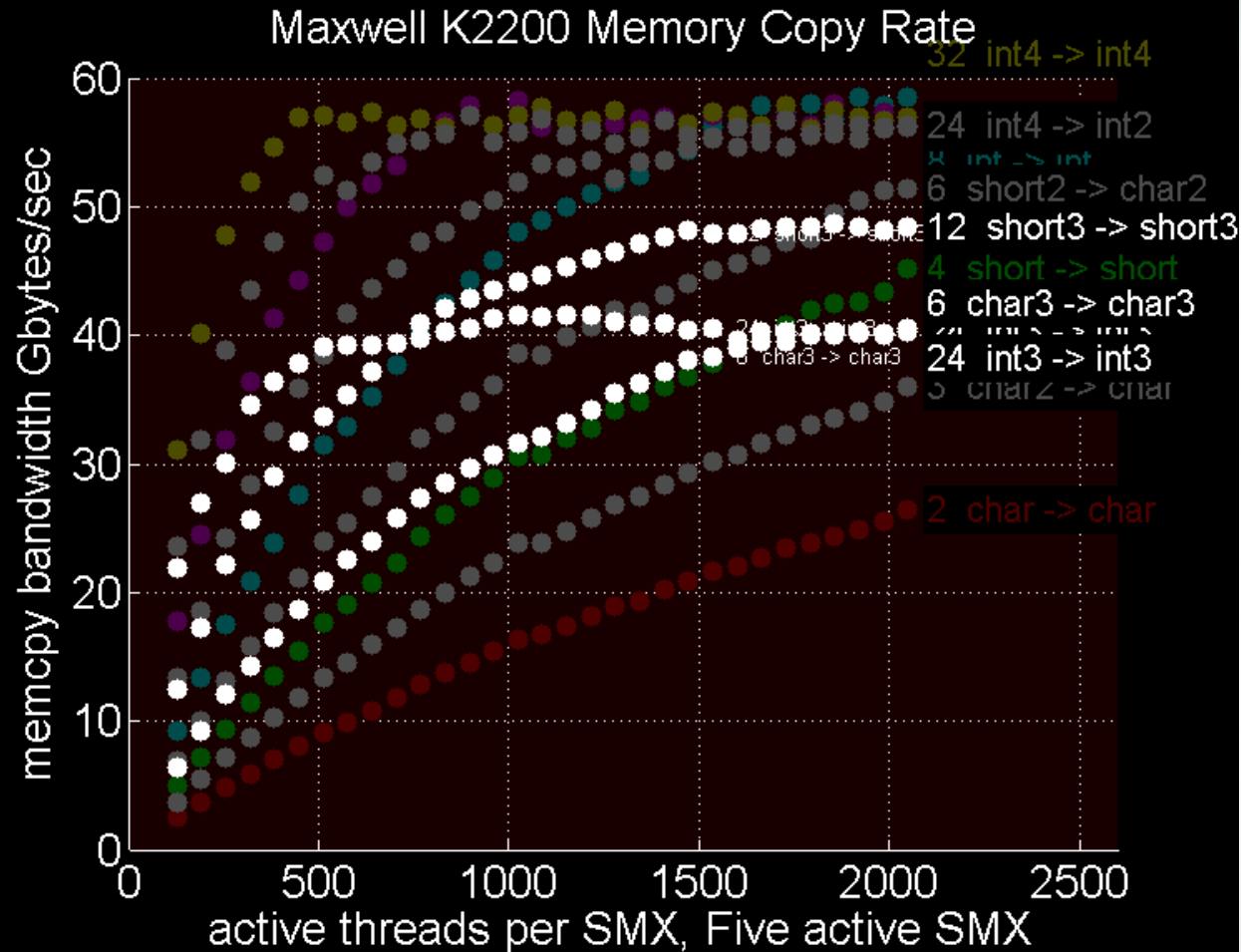
## Kepler K2000



Non-power  
of two work  
per thread  
has a serious  
performance  
penalty in Kepler

This dropoff not seen  
in Fermi Quadro 2000

## Maxwell K2200



Non-power  
of two work  
per thread  
has a smaller  
penalty in Maxwell

# Conclusions on Memory Bandwidth

- Maxwell K2220 kernel-based memory copy throughput is nearly  $1.1\times$  Kepler K2000
- Maxwell requires fewer active threads to saturate the memory bandwidth

# Outcome of the Migration to Maxwell

- Recompile for Compute Capability 5.0 to avoid online JIT compile!
- Much easier than the migration from Fermi to Kepler
- Maxwell easier to achieve the maximum available performance without big code changes:  
**At least 1.1 to 2× better, depending on the degree of I/O or compute in each kernel**



# Thank You for Your Attention and Questions!



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

685 E. Middlefield Road  
Mountain View, CA 94043  
Phone: +1 (650) 969-9112  
[ismayil.guracar@siemens.com](mailto:ismayil.guracar@siemens.com)