

Experiences Porting Real Time Signal Processing Pipeline CUDA Kernels to Kepler and Windows 8

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

GTC2014: S4148 Wednesday 10:00 am

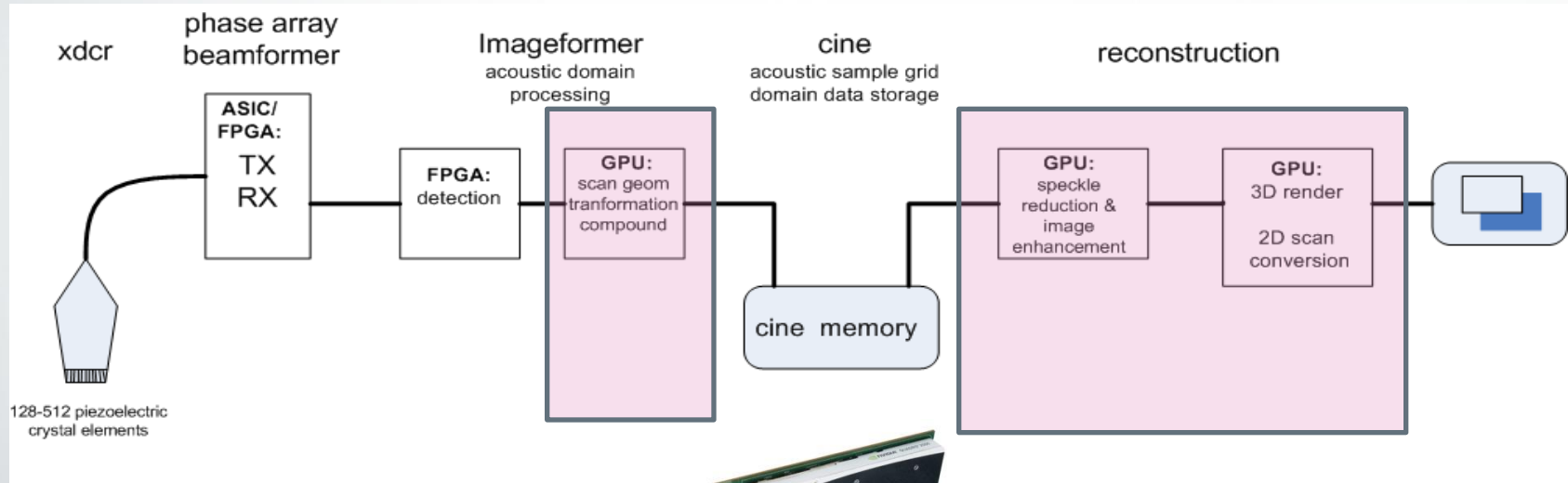
Diagnostic Ultrasound Imaging Equipment



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



ACUSON SC2000™ Ultrasound System Signal Processing Timeline



ACUSON SC2000 Instrument Programming and Hardware Environment

Ultrasound Platform SC2000 1.0 developed in 2008 using WinXP, CUDA 2.3 and originally GeForce 9800GT and a few years later replaced with Fermi Quadro 2000

This talk will be on the migration to Windows 8, CUDA 5.5 and Quadro Kepler K2000

Application #1 2D Speckle Reduction

without



with



2D cross-sectional image of the heart: left ventricle and mitral valve

Application #2 2D Spatial Compounding

without



with

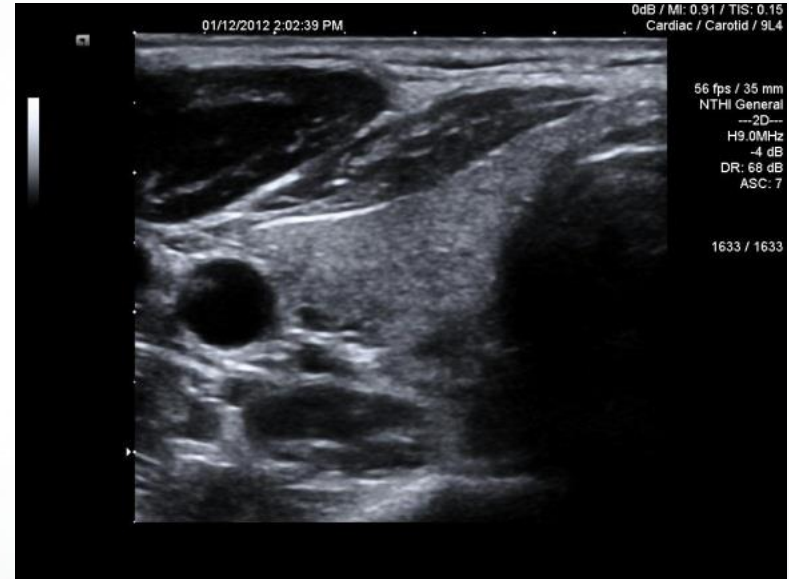
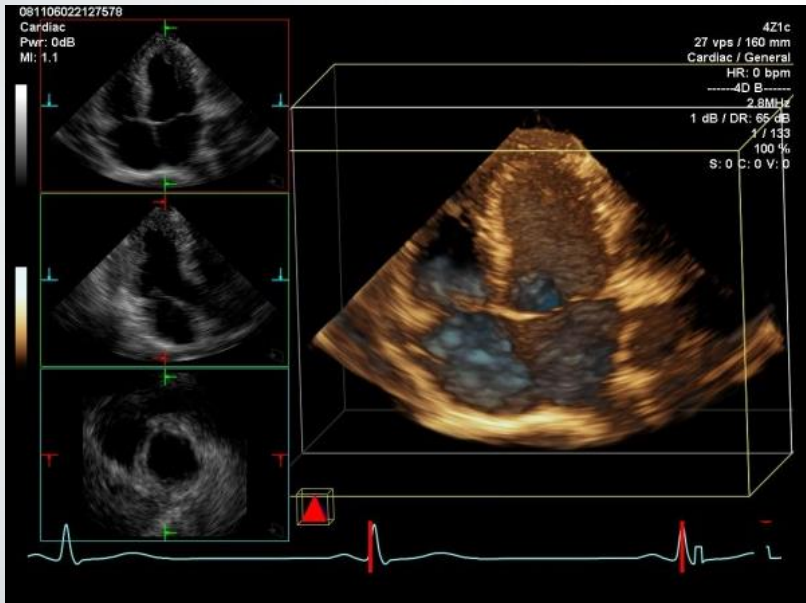


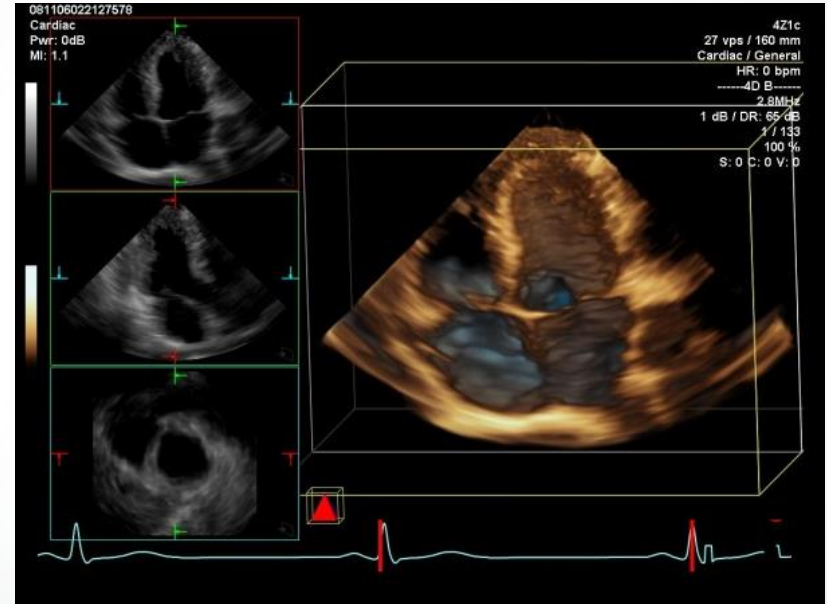
Image of my thyroid gland and cross section of internal carotid artery

Application #3 3D Speckle Reduction

without



with



3D image of the heart, with accompanying 2D orthogonal slices

A First Look at Performance

Just plug in a Kepler K2000 and boot with Win8

Relative processing rate (selected imaging conditions)		
	WinXP Fermi Quadro 2000	Win8 Kepler K2000
Application #1 2D Speckle Reduction	100%	62%
Application #2 2D Spatial Compound	100%	89%
Application #3 3D Speckle Reduction	100%	57%

A Search For Causes and Solutions

- Kernel launch overhead
 - Windows Display Driver Model new for Win7/8
- Kernel execution speed limiting factors
 - execution latency and throughput
 - memory bandwidth

The Transition from Windows XP and the WDDM in Windows 7/8

WDDM: Windows Display Driver Model

- A layer between the CPU and GPU
- GPU Command queue managed by operating system
- CPU-GPU synchronization overhead

Processing Pipeline Example

```
cudaMemcpy(d_ptr1, inputPtr, dataSize, cudaMemcpyHostToDevice);  
  
kernel_1<<<gridSz, blockSz>>>(d_ptr1,d_ptr2,arg1,arg2);  
  
kernel_2<<<gridSz, blockSz>>>(d_ptr2,d_ptr3,arg1,arg2);  
  
kernel_3<<<gridSz, blockSz>>>(d_ptr3,d_ptr4,arg1,arg2);  
  
cudaMemcpy(h_outputPtr, d_ptr4, dataSize, cudaMemcpyDeviceToHost);
```

Processing Pipeline Example: Synchronization for Error Localization

```
cudaMemcpy(d_ptr1, inputPtr, dataSize, cudaMemcpyHostToDevice);
if (cudaGetLastError()) {handleError("upload to GPU error");}




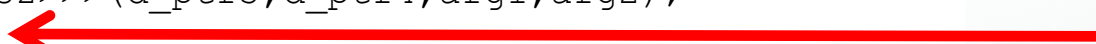
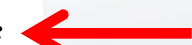
kernel_1<<<gridSz, blockSz>>>(d_ptr1,d_ptr2,arg1,arg2);
cudaDeviceSynchronize();
if (cudaGetLastError()) {handleError("kernel_1 error");}

kernel_2<<<gridSz, blockSz>>>(d_ptr2,d_ptr3,arg1,arg2);
cudaDeviceSynchronize();
if (cudaGetLastError()) {handleError("kernel_2 error");}

kernel_3<<<gridSz, blockSz>>>(d_ptr3,d_ptr4,arg1,arg2);
cudaDeviceSynchronize();
if (cudaGetLastError()) {handleError("kernel_3 error");}

cudaMemcpy(h_outputPtr, d_ptr4, dataSize, cudaMemcpyDeviceToHost);
if (cudaGetLastError()) {handleError("download from GPU error");}
```

Location of Windows 8 Kernel “Thunks” In a Processing Pipeline

```
cudaMemcpy(d_ptr1, inputPtr, dataSize, cudaMemcpyHostToDevice);  “kernel thunks”  
if (cudaGetLastError()) {handleError(“upload to GPU error”);}  
  
kernel_1<<<gridSz, blockSz>>>(d_ptr1,d_ptr2,arg1,arg2);  
cudaDeviceSynchronize();   
if (cudaGetLastError()) {handleError(“kernel_1 error”);}  
  
kernel_2<<<gridSz, blockSz>>>(d_ptr2,d_ptr3,arg1,arg2);  
cudaDeviceSynchronize();   
if (cudaGetLastError()) {handleError(“kernel_2 error”);}  
  
kernel_3<<<gridSz, blockSz>>>(d_ptr3,d_ptr4,arg1,arg2);  
cudaDeviceSynchronize();   
if (cudaGetLastError()) {handleError(“kernel_3 error”);}  
  
cudaMemcpy(h_outputPtr, d_ptr4, dataSize, cudaMemcpyDeviceToHost);   
if (cudaGetLastError()) {handleError(“download from GPU error”);}
```

Experiment: The High Cost of “Thinking”

Sync after each kernel launch

```
for (iter=0; iter<1000; iter++)  
{  
    kernel_1<<<gridSz,blockSz>>>(d_ptr1,...  
    cudaDeviceSynchronize();  
}
```

Sync only once, after 1000 kernel launches

```
for (iter=0; iter<1000; iter++)  
{  
    kernel_1<<<gridSz,blockSz>>>(d_ptr1,...  
}  
cudaDeviceSynchronize();
```

Experiment: The High Cost of “Thinking”

Measurement result by timing each fragment:

With a kernel taking about 600 microseconds to execute the synchronization “think” added about 130 microseconds

Test Platform:

Win7 HP Z620 PC

K2000 graphics card

NVIDIA video Driver 331.65

Your results may differ – try it!

Remove Synchronization and Error Check Between Each Kernel Launch

```
cudaMemcpyAsync(d_ptr1, inputPtr, dataSize, cudaMemcpyHostToDevice, streamId);  
kernel_1<<< gridSize, blockSize, 0, streamId >>>(d_ptr1,d_ptr2,arg1,arg2);  
kernel_2<<< gridSize, blockSize, 0, streamId >>>(d_ptr2,d_ptr3,arg1,arg2);  
kernel_3<<< gridSize, blockSize, 0, streamId >>>(d_ptr3,d_ptr4,arg1,arg2);  
cudaMemcpyAsync(h_outputPtr, d_ptr4, dataSize, cudaMemcpyDeviceToHost, streamId);  
cudaDeviceSynchronize(); ← Single “kernel thunk”  
if (cudaGetLastError()) {handleError(“Error somewhere in pipeline—good luck”);}
```


Why would I want to localize errors to a kernel?

- Development testing: quickly get to the root of a problem
- Deployment: field failure tracking and statistics
 - MTBF important in medical imaging instruments, particularly when used for interventional procedures such as catheter guidance
 - Older gamer grade GPU hardware (GeForce) had rare recurrent hardware failures. Tracked to particular memory access patterns
 - Current hardware (Fermi and Kepler) is very reliable
 - **Recommend workstation grade cards for medical instruments**

Alternative Solutions

- Use synchronization and error checks in debug mode executables and fewer checks for release mode
- TCC (Tesla Compute Cluster) Mode
 - Bypass WDDM and return to WinXP style tightly coupled CPU/GPU interface
 - Use original synchronization model and check for errors after each kernel launch: small overhead to `cudaDeviceSynchronize()`
 - However, can't use the GPU card in TCC mode to drive a display: multiple GPU cards would be painful in our embedded application

Instruction Level Parallelism

Experiments: Kepler versus Fermi

To learn more about ILP, see

Volkov, “Better Performance at Lower Occupancy”

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

Specifications Comparison: Multiprocessor

Fermi Quadro 2000 SM count = 4
GPU clock rate = 626 MHz

Kepler K2000 SMX count = 2
GPU clock rate = 952 MHz

$$4 \times 626 > 2 \times 952$$

Is this a step backward or is SMX > SM?

```
#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;
}
```

← Loops contain purely
computation -- no I/O

```
__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;
}
```

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

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

    #pragma unroll
    for (int y=0; i<N_ITERATIONS; y++)
    {
        a = a*b + c;
        d = d*e + f;
        g = g*h + i;
    }
    ... complete loop & output a,d and g
}
```



3-way ILP

```
__global__ void
ilp4_kernel(float *d_In, float *d_Out)
{
    ... initialize variables, setup loop

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

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

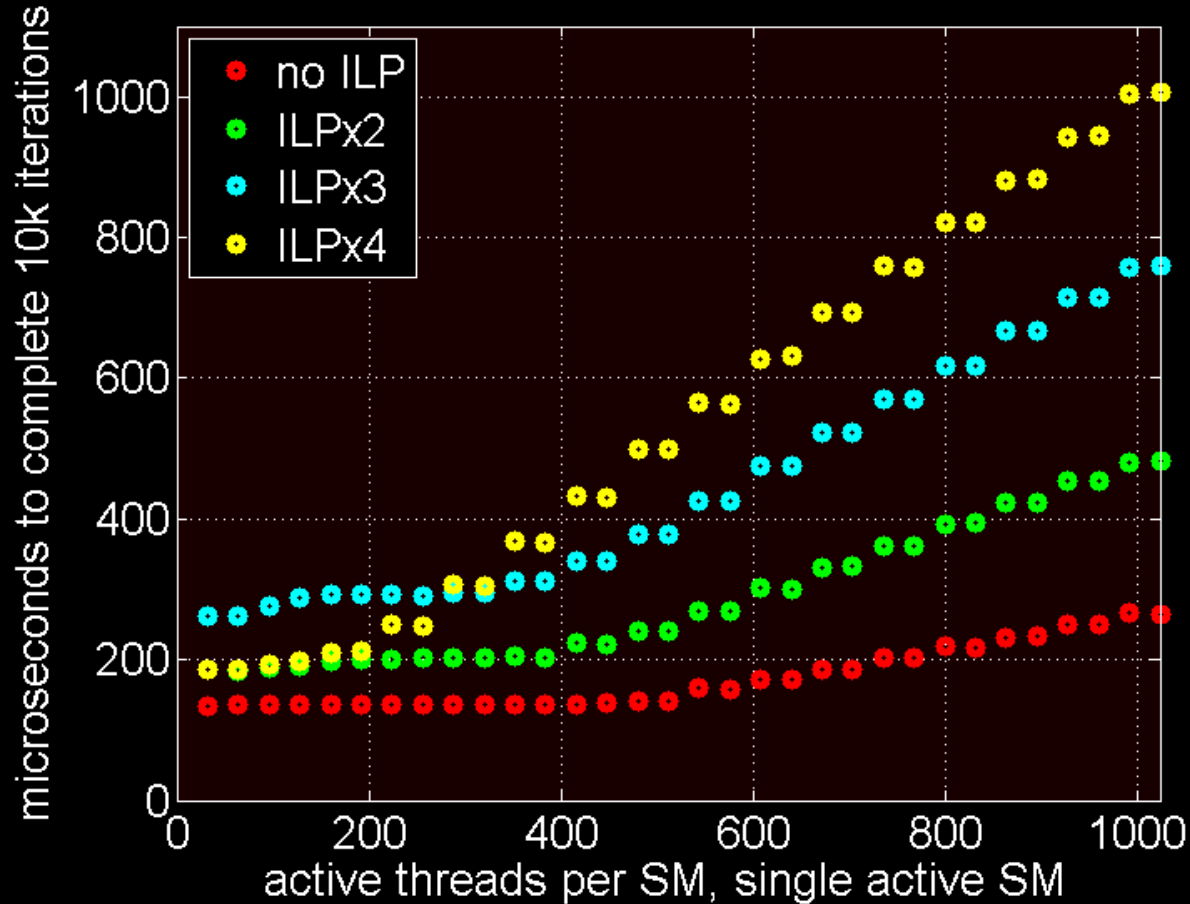


4-way ILP

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);  
}
```

Fermi Quadro 2000

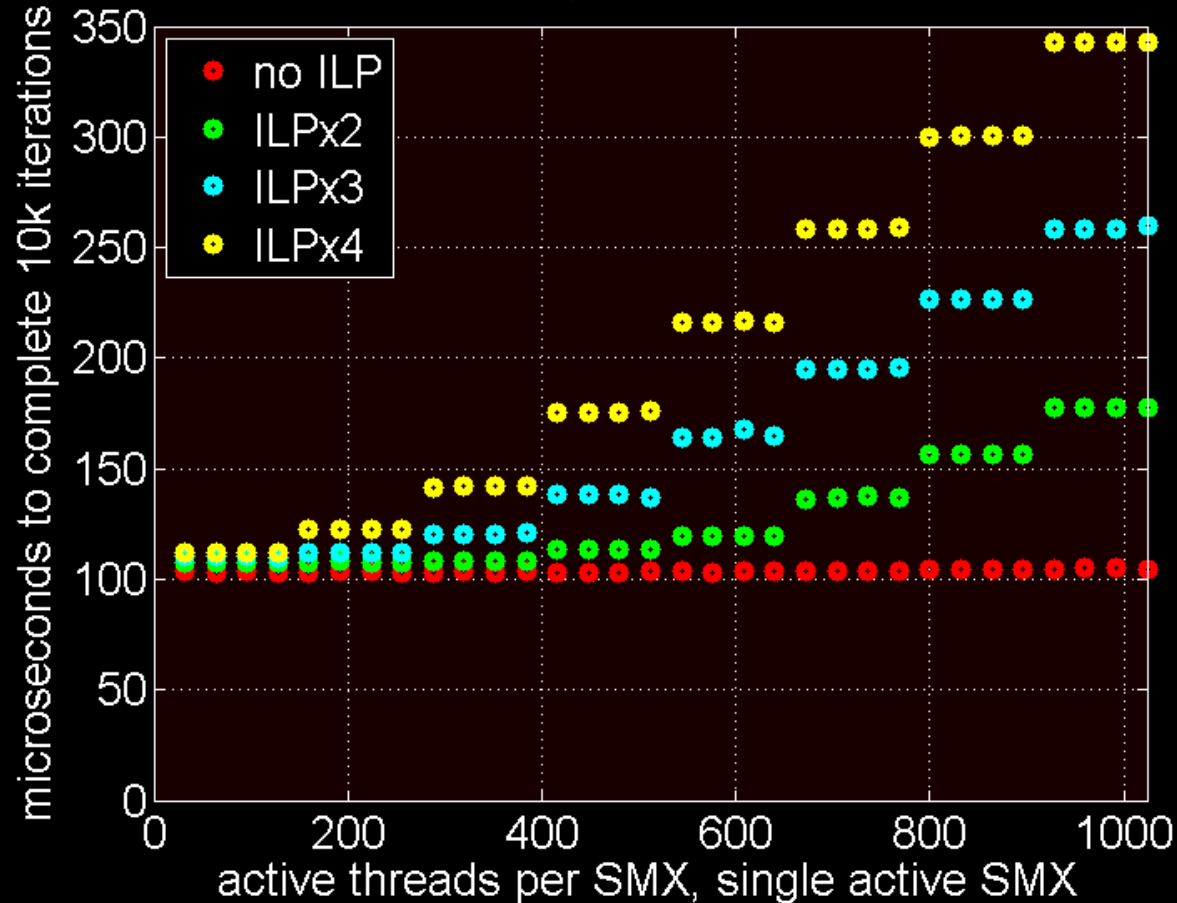


Fermi Quadro 2000

Processing latency made visible--additional active threads produce no increase in execution time

64-thread steps due to 2 warp schedulers
(Compute Capability 2.1)

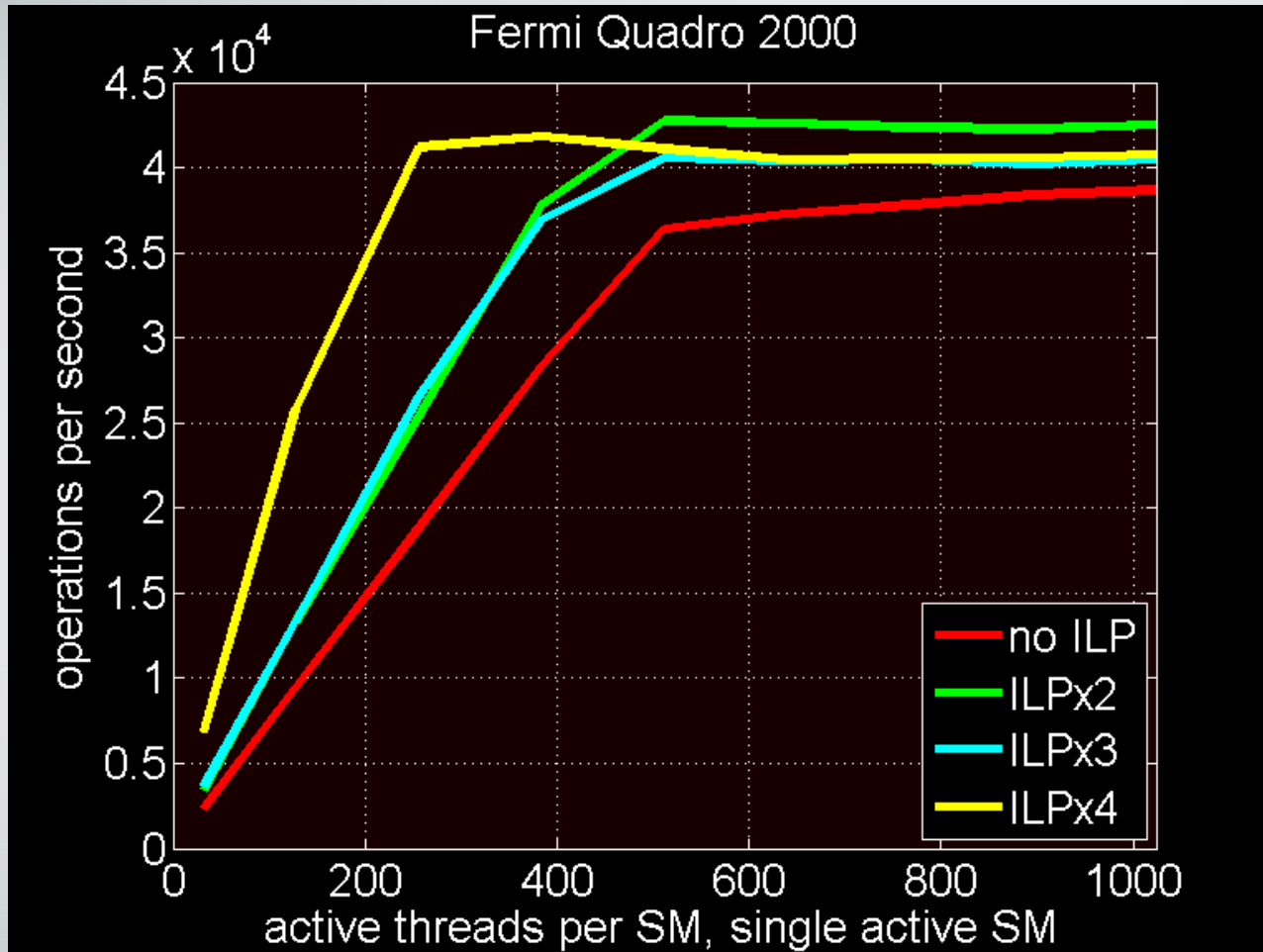
Kepler K2000

**Kepler K2000**

With no ILP additional active threads cause no increase in execution

Not enough work to hide the execution latency even with 1024 threads

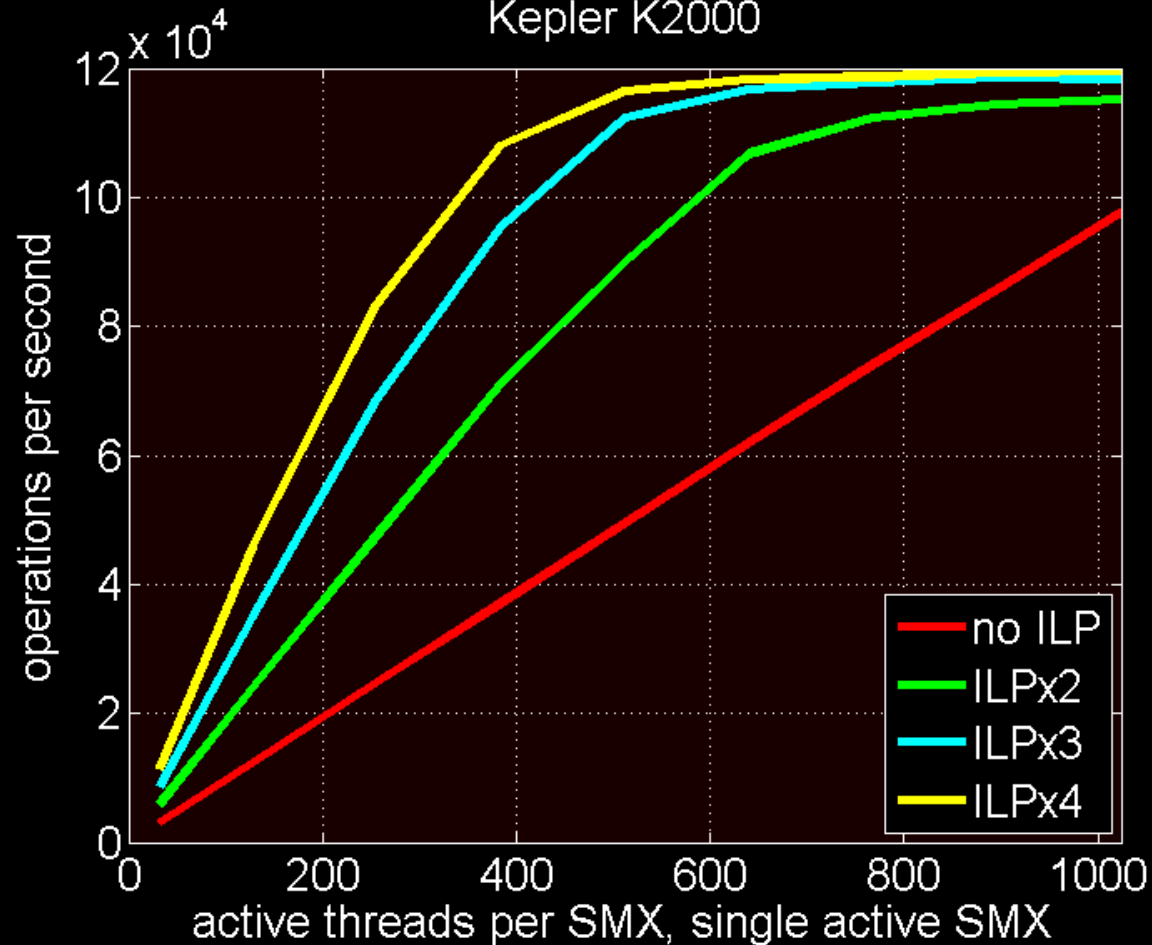
128-thread wide steps due to 4 warp schedulers (Compute Capability 3.0)



Fermi Quadro 2000

Execution throughput saturates with fewer active threads per SM with ILP

Kepler K2000



Kepler K2000

ILP is **required** to provide enough work to hide execution latency

Even with lots of ILP still need about 512 active threads per SM to saturate execution units

ILP factor	Active Threads Per SM/SMX	Q2000 Fermi ×10 ⁴ op/sec Per SM	K2000 Kepler ×10 ⁴ op/sec Per SMX	Relative performance Kepler op/sec × 2 SMX ÷ Fermi op/sec × 4 SM
no ILP	256	1.88	2.48	0.66 ×
	512	3.63	4.94	0.75 ×
	768	3.78	7.41	0.98 ×
	1024	3.86	9.77	1.27 ×
ILP×2	256	2.53	4.76	0.94 ×
	512	4.27	8.99	1.05 ×
	768	4.23	11.23	1.32 ×
	1024	4.24	11.51	1.36 ×
ILP×4	256	4.11	8.3	1.01 ×
	512	4.11	11.6	1.41 ×
	768	4.11	11.8	1.44 ×
	1024	4.07	11.95	1.46 ×

Conclusions on the Multiprocessor and ILP

Kepler SMX is a lot more capable than Fermi SM
2 Kepler SMX clocked at 952 MHz
~ **1.4x more powerful**
than 4 Fermi SM clocked at 625 MHz

Require ILP to realize the full potential of Kepler

Also need to keep occupancy up (>50% is better)

Memory Bandwidth Experiments: R/W bytes per thread Kepler versus Fermi

How does a purely I/O bound task scale?

Specifications Comparison: Memory Bandwidth

nominal memory bandwidth

Fermi Quadro 2000

41.7 Gbytes/sec

Kepler K2000

64 Gbytes/sec

For an I/O bound task $K2000 > Q2000$
Is this always true?

Kernel Based Memory Copy Experiment

Varying amounts of 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];
}

extern "C"
void mem2(char *d_in, char *d_out, int bytesToCopy, int threadCount,
          int sharedMemPerThreadBlock)
{
    int blocks = bytesToCopy/(2*threadCount);
    int pitch=threadCount;
    dim3 gridSz(blocks);
    dim3 blockSz(threadCount);
    mem2_kernel<<<gridSz,blockSz, sharedMemPerThreadBlock>>>(
        (short*)d_in, (short*)d_out, pitch);
}
```

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];
}

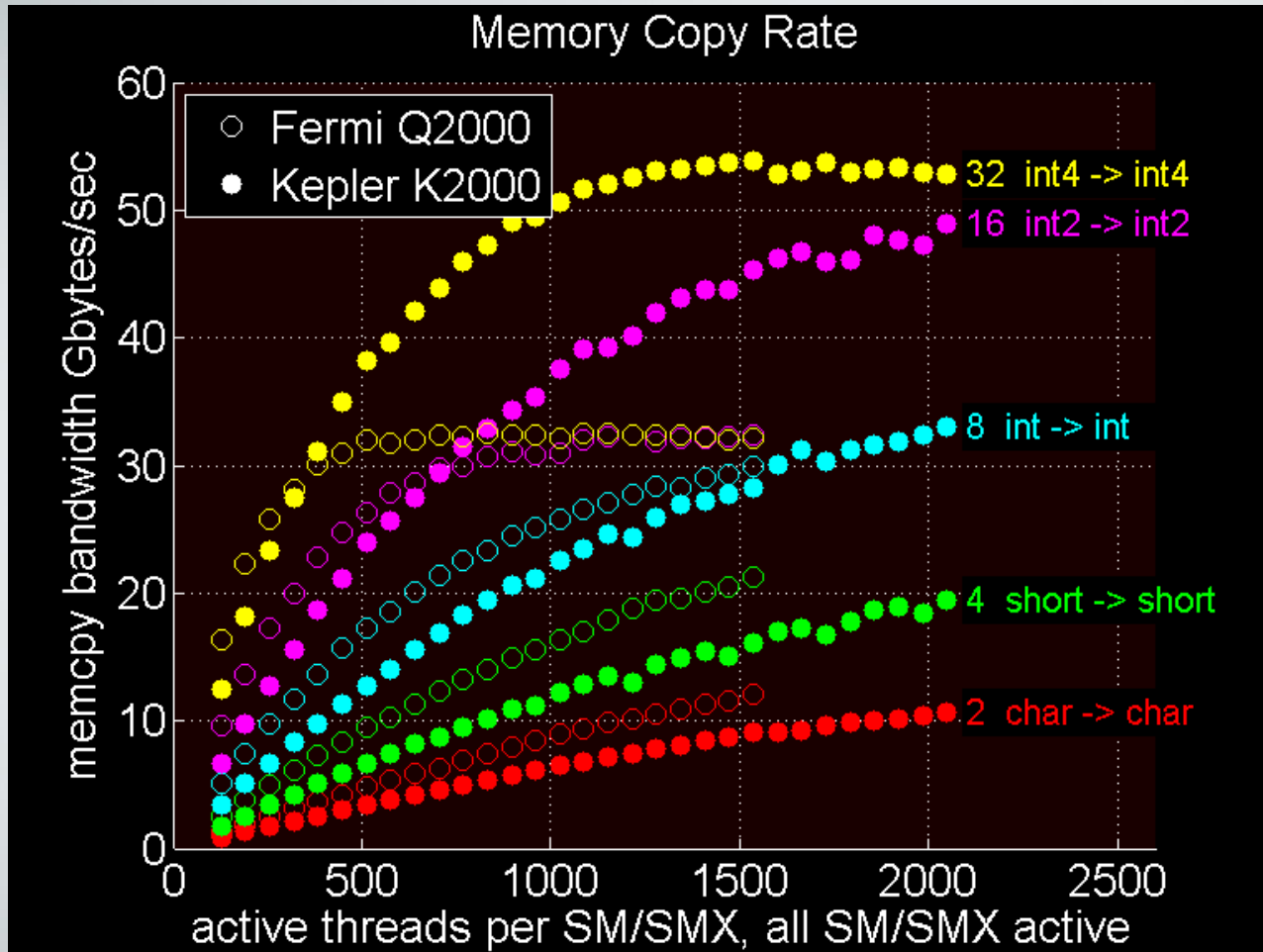
extern "C"
void mem3(char *d_in, char *d_out, int bytesToCopy, int threadCount,
          int sharedMemPerThreadBlock)
{
    int blocks = bytesToCopy/(3*threadCount);
    int pitch=threadCount;
    dim3 gridSz(blocks);
    dim3 blockSz(threadCount);
    mem3_kernel<<<gridSz,blockSz, sharedMemPerThreadBlock>>>(
        (char3*)d_in, (char3*)d_out, pitch);
}
```

Read int2 / Write short2 8 bytes read/4 bytes write

```
__global__ void
mem8to4_kernel(int2 *d_In, short2 *d_Out, int pitch)
{
    int ii = threadIdx.x + blockIdx.x * pitch;
    int2 V = d_In[ii];
    d_Out[ii]=make_short2(V.x,V.y);
}
```

**I/O is 12 bytes
per kernel**

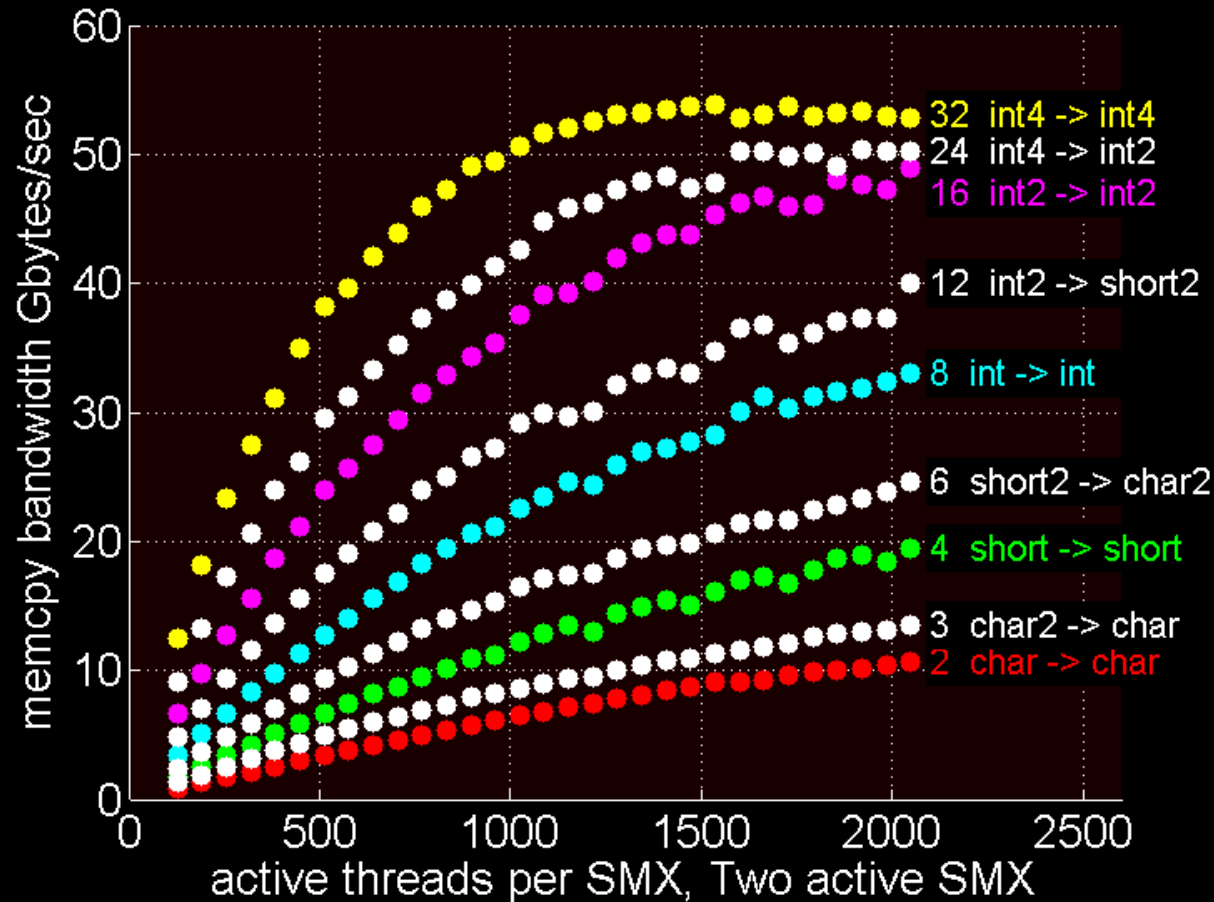
```
extern "C"
void mem8To4(char *d_in, char *d_out, int bytesToCopy,
             int threadCount, int sharedMemPerThreadBlock)
{
    int blocks = bytesToCopy/(8*threadCount);
    int pitch=threadCount;
    dim3 gridSz(blocks);
    dim3 blockSz(threadCount);
    mem8to4_kernel<<<gridSz,blockSz, sharedMemPerThreadBlock>>>(
        (int2*)d_in, (short2*)d_out, pitch);
}
```



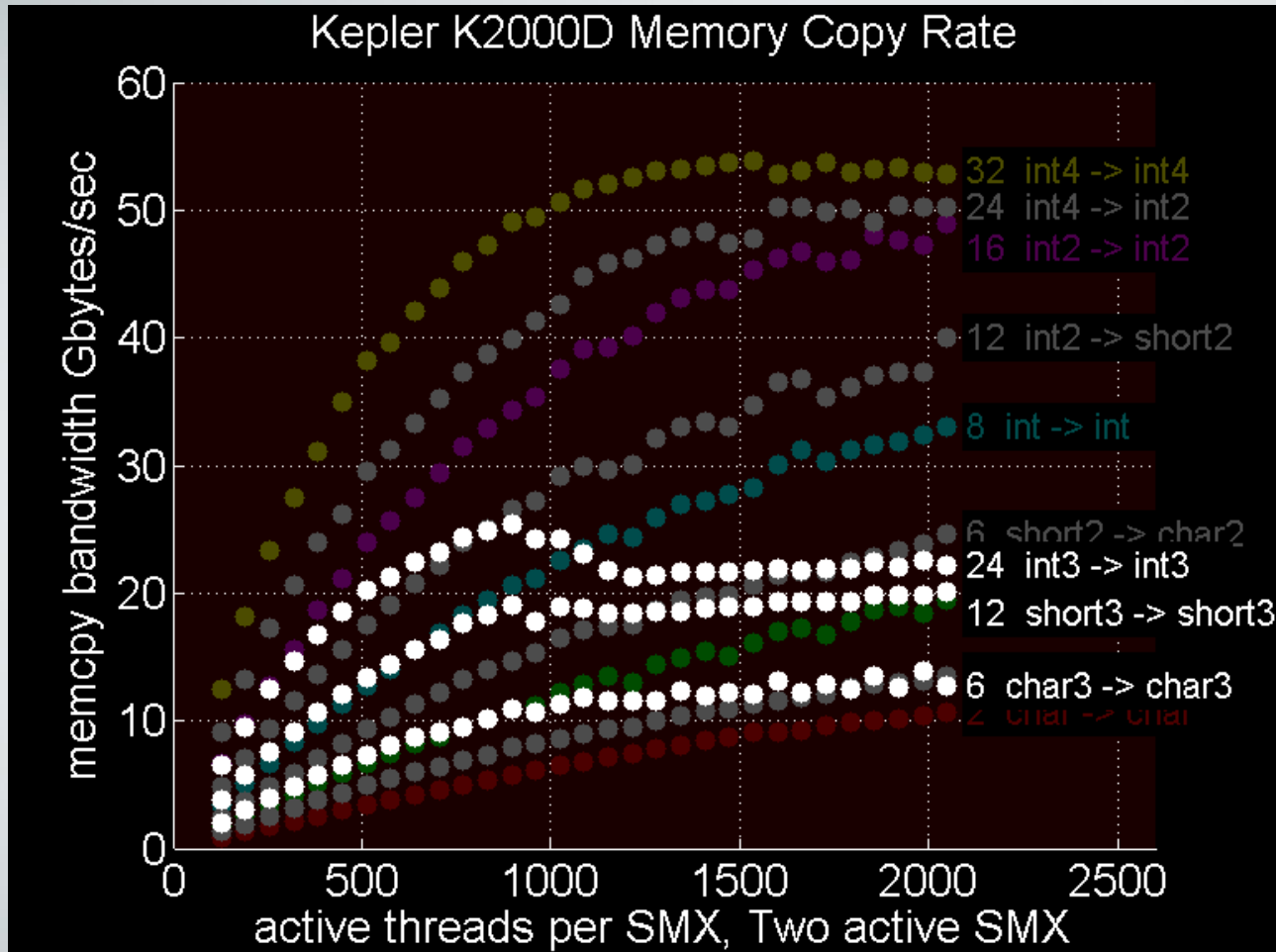
In many situations
Fermi Quadro 2000
memory bandwidth
is greater than
Kepler K2000

Max active threads
per SM/SMX
CC 2.1 = 1536
CC 3.0 = 2048

Kepler K2000D Memory Copy Rate



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



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

**This dropoff not seen
in Fermi Quadro 2000**

Conclusions on Memory Bandwidth

For a purely I/O bound task to saturate device memory K2000 needs at least 50% occupancy (1024 active threads per SMX) and each thread reading/writing about 32 bytes (i.e. read int4 → write int4)

100% occupancy and 16 bytes/thread will get close to saturating memory

Seems to be a very large penalty in Kepler for non power-of-two bytes per thread read/write access

Outcome of the Migration

Mitigated WDDM kernel launch overhead

Rewrote key kernels for greater ILP and more memory access work per thread

Refactored the existing code for all three applications and exceed the performance requirements with Kepler K2000 and Windows 8.

It is possible to take advantage of the advances in technology that Kepler brings – it just takes a little work!

Prediction

Pipelines will probably get even longer in the future, requiring more active threads with greater amounts of ILP to maximize computational throughput and memory bandwidth

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: (650) 969-9112
ismayil.guracar@siemens.com