Lecture 8

Matrix Multiplication Using Shared Memory

Announcements

- A2 Due
- Friday’s makeup lecture 4:00 to 5:20pm in EBU3B 2154
- Next Tuesday’s lecture: be prepared to discuss the assigned paper in class: “Debunking to 100X GPU vs CPU myth …

Project Proposals

- Due 2/24
- What are the goals of your project? Are they realistic?
- What are your hypotheses?
- What is your experimental method for proving or disproving your hypotheses?
- What experimental result(s) do you need to demonstrate?
- What would be the significance of those results?
- What code will you need to implement? What software packages or previously written software will use?
- A tentative division of labor among the team members
- A preliminary list of milestones—with completion dates

Today's lecture

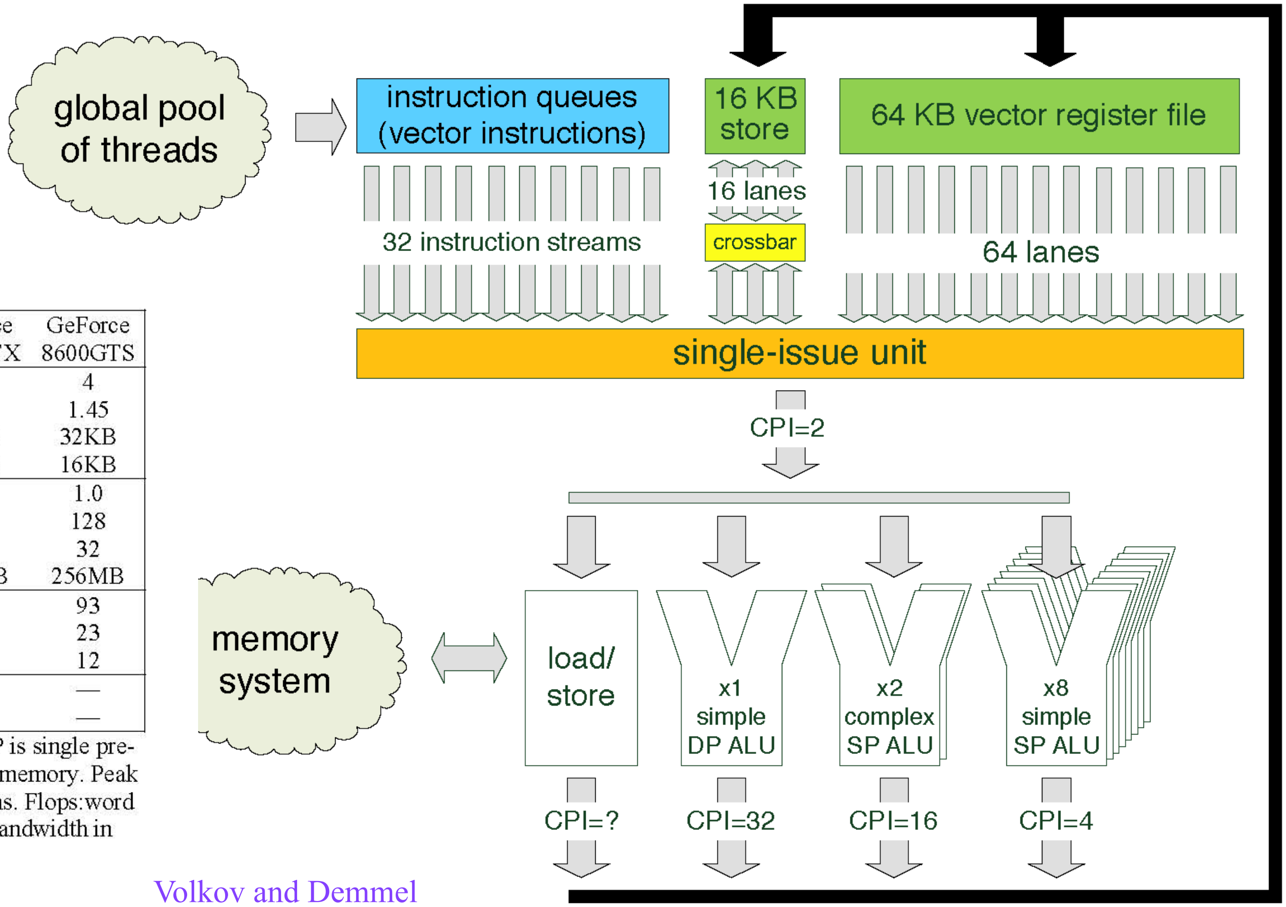
- Matrix Multiplication with Global Memory
- Using Shared Memory – part I

Recapping from last time

Many Multithreaded Vector Units

GPU name	GeForce GTX280	GeForce 9800GTX	GeForce 8800GTX	GeForce 8600GTS
# of vector cores	30	16	16	4
core clock, GHz	1.30	1.67	1.35	1.45
registers/core	64KB	32KB	32KB	32KB
smem/core	16KB	16KB	16KB	16KB
memory bus, GHz	1.1	1.1	0.9	1.0
memory bus, pins	512	256	384	128
bandwidth, GB/s	141	70	86	32
memory amount	1GB	512MB	768MB	256MB
SP, peak Gflop/s	624	429	346	93
SP, peak per core	21	27	22	23
SP, flops:word	18	25	16	12
DP, peak Gflop/s	78	—	—	—
DP, flops:word	4.4	—	—	—

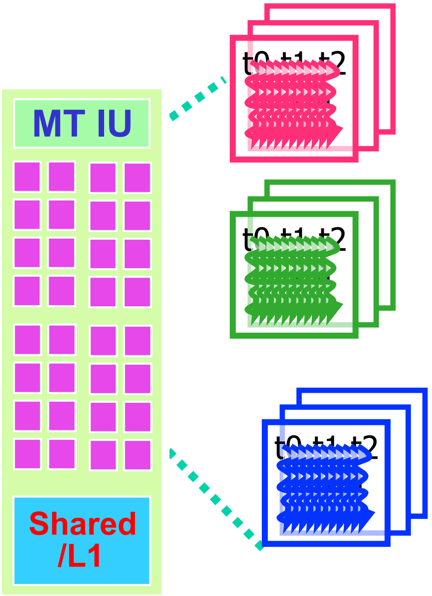
Table 1: The list of the GPUs used in this study. SP is single precision and DP is double precision. Smem is shared memory. Peak flop rates are shown for multiply and add operations. Flops:word is the ratio of peak Gflop/s rate to pin-memory bandwidth in words.

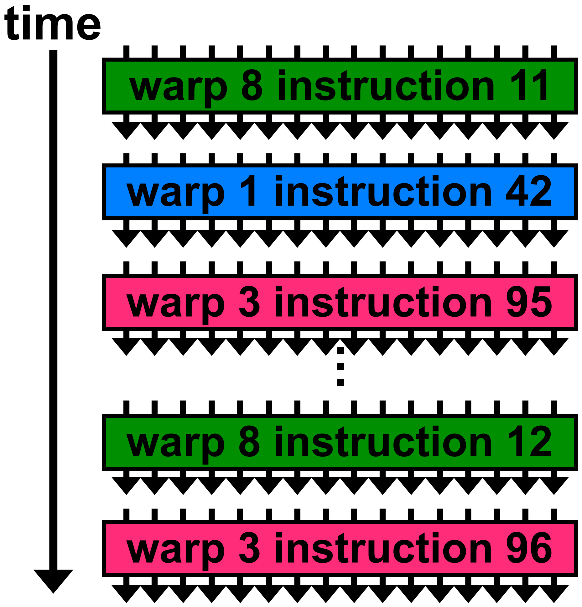


Volkov and Demmel

Warp scheduling on Fermi

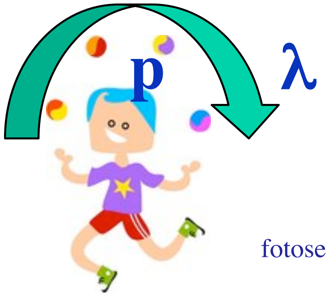
- Assign threads to an SM in units of a thread block
- Hardware is free to assign blocks to any SM, multiple blocks per SM
- Blocks are divided into *warps* of 32 (SIMD) threads, a schedulable unit
    - Dynamic instruction reordering
    - All threads in a Warp execute the same instruction
    - Multiple warps simultaneously active, hiding data transfer delays
    - All registers in all the warps are available, 0 overhead scheduling





Occupancy

- A minimum number of warps needed to hide memory latency
- Occupancy: # active warps ÷ max # warps supported by vector unit
- Limited by vector unit resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of threads
- Consider a kernel (16x16 block size)
  - Shared memory/block = 2648 bytes
  - Reg/thread=38 [38*256 =9728 < 16k]
  - # available registers is the limiting factor



fotosearch.com

- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption

Data motion cost

- Communication performance is a major factor in determining the overall performance of an application
- The $\alpha-\beta$ model: $\alpha+\beta^{-1}_{\infty}\ n$

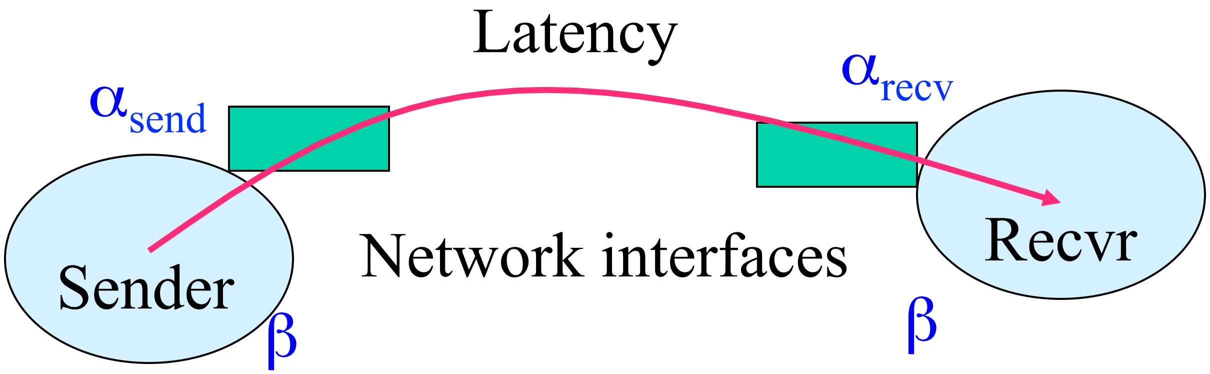
n = message length

α = message startup time

β_{∞} = peak bandwidth (bytes / second)

Machine	β_{∞} (Dev)	H-D	D-H
Forge	103 GB/s	2.3	1.3
Lilliput	73.6	3.3	2.8
CseClass01	122	3.4	2.7
CseClass04	56.1	5.2	4.1

As reported by bandwidthTest



Consequences of data motion cost

- Consider saxpy z[i] = a*x[i]+y[i]
- This is a bandwidth bound kernel
- Running time under the $\alpha-\beta$ model: $\alpha+\beta^{-1}_{\infty}$ n

  α = 4 µs

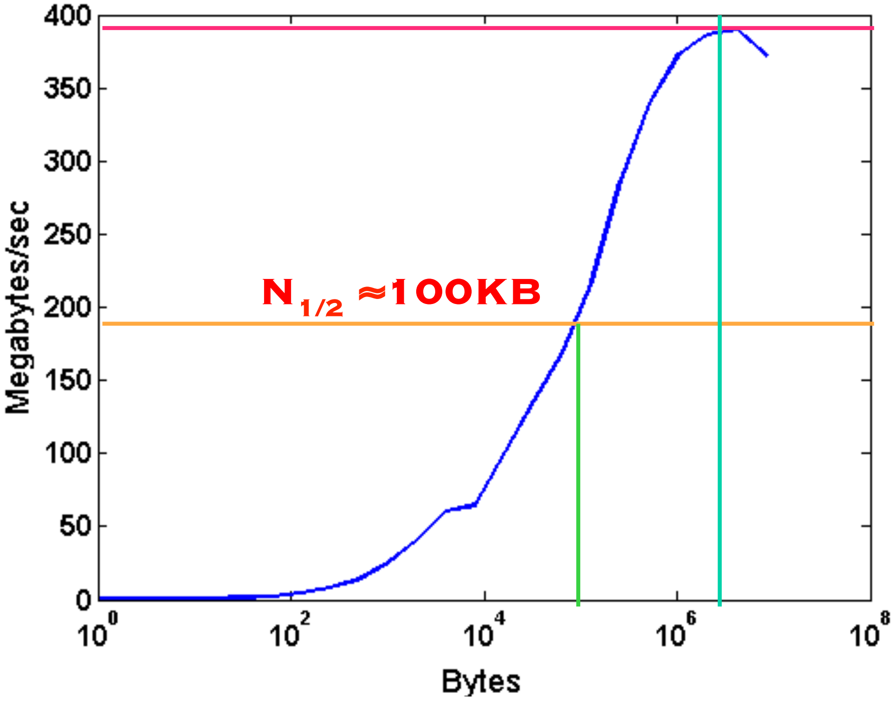
  β_{∞} = 127 GB/sec
- Flop rate bounded by (2n flops/12n bytes)* β_{∞}
  - (1/6) flops per byte of bandwidth: 27 Gflops/sec
- $N_{1/2}$ half power (bandwidth) point: N ≈ 42,000
  - The transfer size required to achieve ½ β_{∞}
  - Matrix multiplication takes 4 µs
  - But the largest matrix that fits into memory is 1GB ~ $(16K)^2$
  - Consequence: saxpy takes constant time to run

Half power point

- We define the ***half power point*** $n_{1/2}$ as the transfer size required to achieve ½ β_∞

  $$\tfrac{1}{2}\,\beta^{-1}_\infty = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \tfrac{1}{2}\,\beta^{-1}_\infty$$

- In theory, this occurs when $\alpha = \beta^{-1}_\infty\, n_{1/2} \Rightarrow n_{1/2} = \alpha\beta_\infty$
- Formula may not be accurate



(SDSC Blue Horizon)

Latency

- Instructions waits on dependencies
  x = a + b; // ~20
  y = a + c; // independent
  (stall)
  z = x + d; // dependent
- How many warps are needed to hide latency if minimum latency is 4 cycles per instruction?
- Latencies, can vary but for single precision
  - GT200 (C1060, Lilliput): 24 CP * 8 cores / SM = 192 ops/cycle
  - GF100 (GTX-580, Cseclass01/02): 18 CP * 32 = 576
  - GF104 (GTX 460, Cseclass03-07): 18 CP * 48   = 864
- Measuring latencies
  - a = a*b+c, unrolled, 1 scalar thread on entire device
  - No overflow
  - Hardware not optimized for special values 0,1

Thread vs instruction level parallelism

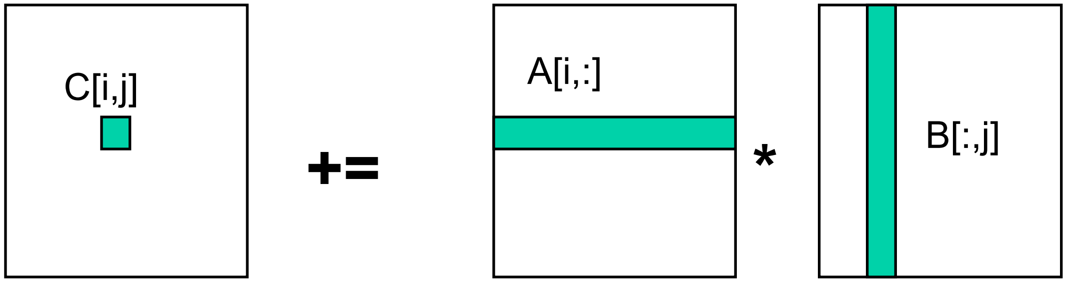
- We are told to maximize the number of threads
- But we can also use instruction level parallelism to boost performance at a lower occupancy
- See Volkov's presentation: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
- On the GF104, we must use ILP to go beyond 66% of peak
  - 48 cores/SM, half warp issues at a time
  - But we have only 2 schedulers
  - We must issue 2 instructions per warp in the same cycle

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ ){
        a = a*b+c;
        d = d * b + c;
}
```

Matrix Multiplication (code in $PUB/Examples/CUDA/MM)

Naïve Host Code

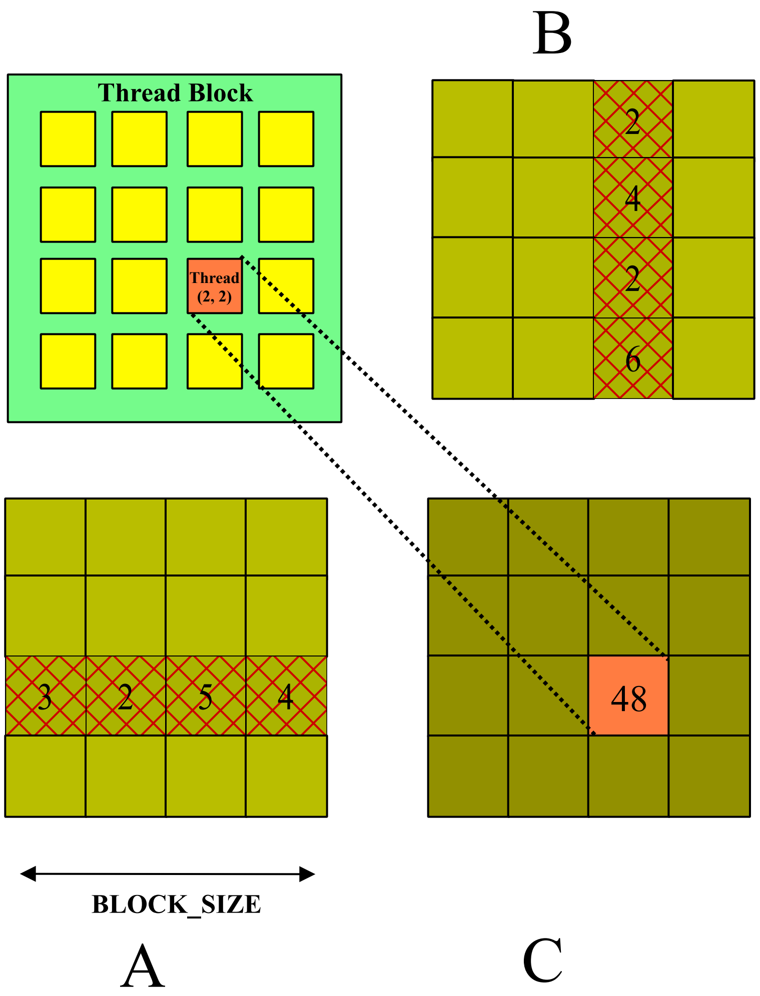
```
// “ijk” kernel
 for i := 0 to n-1
   for j := 0 to n-1
      for k := 0 to n-1
        C[i,j] += A[i,k] * B[k,j]
```



```
for (unsigned int i = 0; i < N; i++)
        for (unsigned int j = 0; j < N; j++) {
            DOUBLE sum = 0;
            for (unsigned int k = 0; k < N; k++)
                sum += A[i * N + k] * B[k * N + j];
            C[i * N + j] = (DOUBLE) sum;
        }
```

Naïve kernel implementation

- Each thread computes one element of C
  - Loads a row of matrix A
  - Loads a column of matrix B
  - Computes a dot product
- Every value of A and B is loaded N times from global memory



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Naïve Kernel

```
__global__ void
matMul(DOUBLE* C, DOUBLE* A, DOUBLE* B) {
 int I  =  blockIdx.x*blockDim.x + threadIdx.x;
 int J =  blockIdx.y*blockDim.y + threadIdx.y;
 int N =  blockDim.y*gridDim.y; // Assume a square matrix
 if ((I < N) && (J < N)){
      float _c = 0;
      for (unsigned int k = 0; k < N; k++) {
          float a = A[I * N + k];
          float b = B[k * N + J];
          _c += a * b;
      }
      C[I * N + J] = _c;
  }
}
```

```
for (unsigned int i = 0; i < N; i++)
      for (unsigned int j = 0; j < N; j++) {
         DOUBLE sum = 0;
         for (unsigned int k = 0; k < N; k++)
            sum += A[i * N + k] * B[k * N + j];
         C[i * N + j] = (DOUBLE) sum;
      }
```

CUDA code on the host side

```
unsigned int n2 = N*N*sizeof(DOUBLE);
DOUBLE *h_A = (DOUBLE*) malloc(n2);
DOUBLE *h_B = (DOUBLE*) malloc(n2);
// Check that allocations went OK
 assert(h_A);  assert(h_B);

genMatrix(h_A, N, N);  genMatrix(h_B, N, N);  // Initialize matrices

DOUBLE *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, n2); ... &d_A  ...  &d_B
checkCUDAError("Error allocating device memory arrays");

 // copy host memory to device
 cudaMemcpy(d_A, h_A, n2, cudaMemcpyHostToDevice);
 cudaMemcpy(d_B, h_B, n2, cudaMemcpyHostToDevice);
 checkCUDAError("Error copying data to device");
```

Host code - continued

```
// setup execution configurations
    dim3 threads(ntx, nty,1);     // ntx & nty are user input
    dim3 grid(n / threads.x, N / threads.y);

     // launch the kernel
     matMul<<< grid, threads >>>(d_C, d_A, d_B);

     // retrieve result
     cudaMemcpy(h_C, d_C, n2, cudaMemcpyDeviceToHost);
     checkCUDAError("Unable to retrieve result from device");

// Free device storage
     assert(cudaSuccess ==cudaFree(d_A));
     assert(cudaSuccess ==cudaFree(d_B));
     assert(cudaSuccess ==cudaFree(d_C));
```

Configuration variables

- Types to manage thread geometries
- dim3 gridDim, blockDim
  - Dimensions of the grid in blocks (gridDim.z not used)
  - Dimensions of a thread block in threads
- dim3 blockIdx, threadIdx;
  - Block index within the grid
  - Thread index within the block

```
__global__ void KernelFunc(...);
dim3   DimGrid(40, 30);    // 1200 thread blocks
dim3   DimBlock(4, 8, 16);   // 512threads per block
Kernel<<< DimGrid, DimBlock, >>>(...);
```

Performance

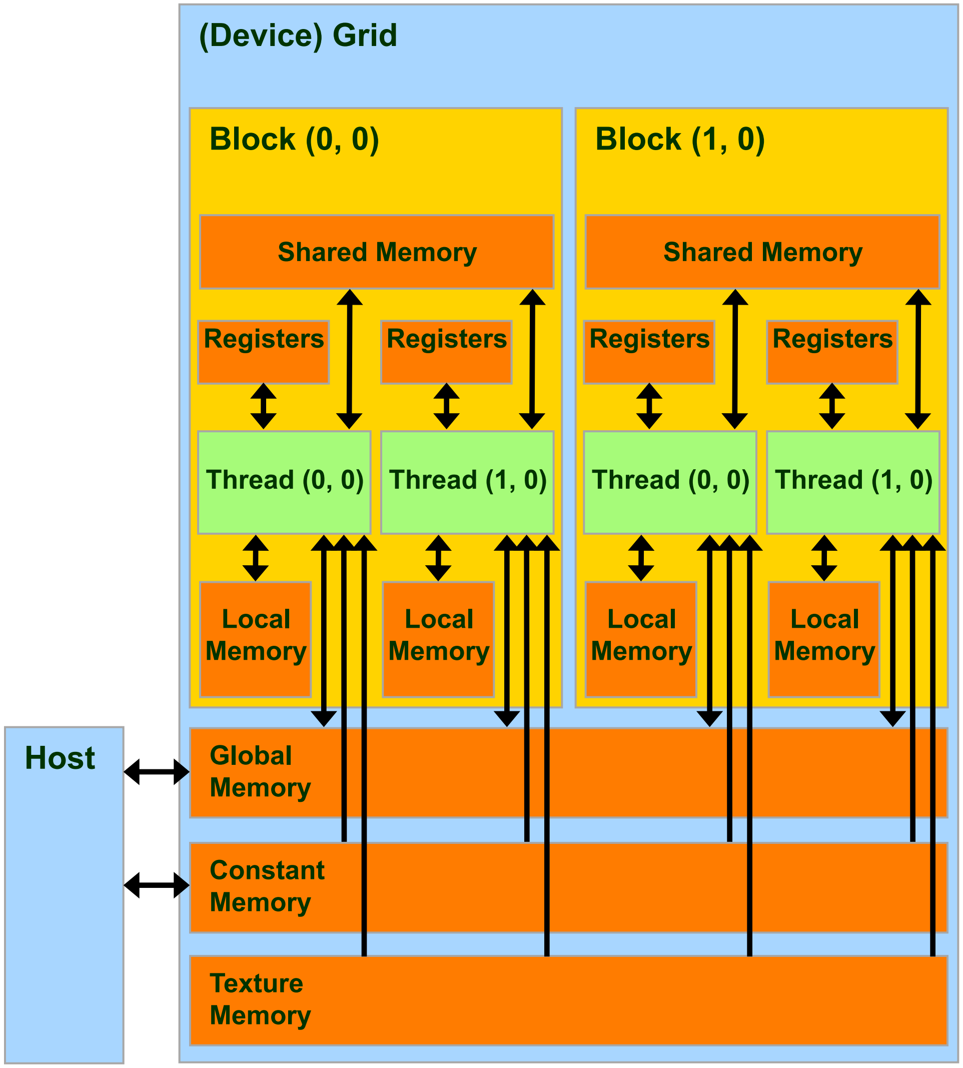
- Baseline [N=512]
  - Lilliput, C1060, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
  - Forge, M2070 14×32 cores
  - 21 GF on 4 CPU cores (MPI), 25 Gflops for N=2K

Gflops dp Lilliput	**9.8**	**8.5**	**7.4**	**5.9**	**5.3**	**5.1**	**3.0**	**2.7**
Geometry	2×256	2×128	2×64	4×128	4×64 2×32	4×32	8×64	8×32

Gflops sp Lilliput	**8.6**	**7.7**	**6.2**	**4.6**	**3.9**	**3.5**	**2.0**	**1.8**
Geometry	2×256	2×128	2×32 2×64	4×128	4×64	4×32	8×64	8×32

Gflops sp Forge dp	**65 48** **64 46**	**56 39**	**52 29** **50 28**	**46 29**	**33 27**	**21**	**8.6**	**6.8** **6.2**
Geometry	2×128 2×256	2×64	4×128 4×64	4×32	2×32 8×64	16×32	32×16	32×8 32×4

Memory Hierarchy



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Name	Latency (cycles)	Cached
Global	DRAM – 100s	No
Local	DRAM – 100s	No
Constant	1s – 10s – 100s	Yes
Texture	1s – 10s – 100s	Yes
Shared	1	--
Register	1	--

Shared Memory/Cache

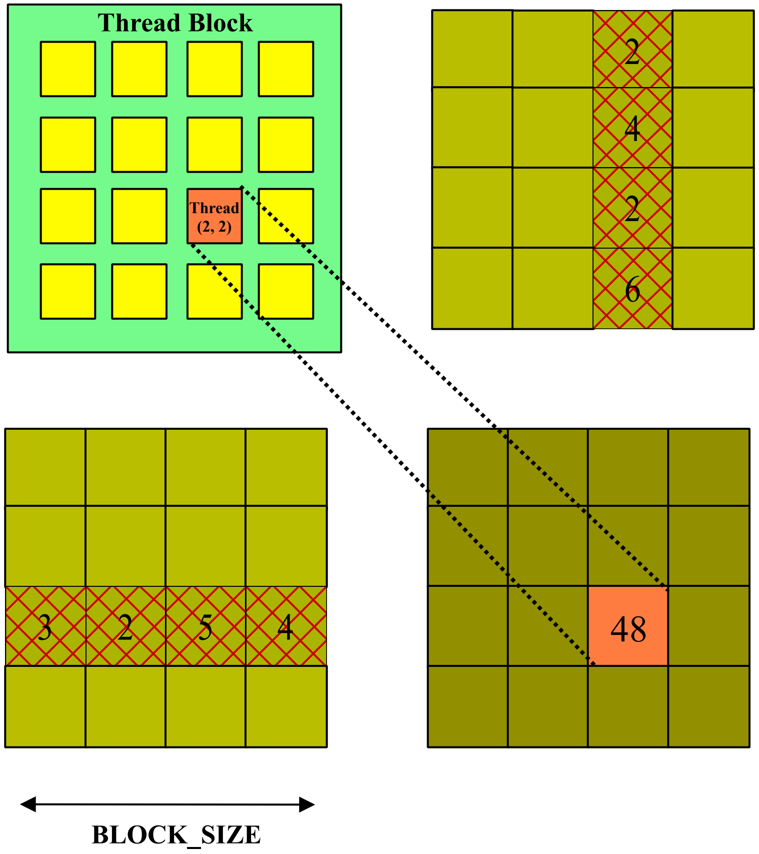
- On-chip local store: pshared memory, partially L1
  - 16KB shared memory + 48 KB L1 cache
  - 48KB shared memory + 16 KB L1 cache
  - 1 for each vector unit
  - All threads in a block share this on-chip memory
    - A collection of warps share a portion of the local store
- Cache accesses to local or global memory, including temporary register spills
- L2 cache shared by all vector units
- Cache inclusion (L1 ⊂ L2?) partially configurable on per-access basis with mem. ref. instruction modifiers
- 128 byte cache line size
- Set the mode using cudaFuncSetCacheConfig()
  cudaFuncSetCacheConfig(boundariesX,PREFERENCE)
  PREFERENCE = {cudaFuncCachePreferShared, cudaFuncCachePreferL1}

A better matrix multiply

- Use shared memory to increase re-use
- Avoid thread divergence
- Memory Coalescing, avoid Bank Conflicts
  - Next time

Improving locality

- Naïve algorithm
  - Each thread loads all the data it needs, independently loads a row and column of input
  - Each input element loaded multiple times
  - Each thread computes 1 MAD + 2 loads + 1 store
- Blocked algorithm
  - Threads cooperate to load a block of A&B into on-chip shared memory
  - Each thread in the block performs the *ijk* loop within shared memory
  - Each thread: *b* mpy-adds + 1 load + 1 store



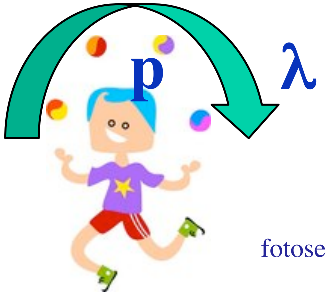
Using shared memory (uncoalesced glbl)

```
__global__ void matMul( float* C, float* A, float* B, int N) {
   const unsigned int bx = BLOCK_X, by = BLOCK_Y;
   const unsigned int tx = threadIdx.x, ty = threadIdx.y;
   const unsigned int I =  blockIdx.x*bx + tx, J =  blockIdx.y*by + ty;
   const unsigned int gx = gridDim.x, gy = gridDim.y;
__shared__ float  a[BLOCK_X][BLOCK_Y], b[BLOCK_X][BLOCK_Y];
   if ((I < N) && (J < N)){
      float c = 0.0f;
      for (unsigned int k=0; k < gy; k++){
         a[tx][ty] = A[ I*N+k*by+ty];
         b[ty][tx] = B[J+N*(k*bx+tx)];
         __syncthreads();       // Synchronizes all threads in a block
         for (unsigned int kk=0; kk< bx; kk++)
            c += a[kk][tx]*b[kk][ty];
         __syncthreads();       // Avoids memory hazards
   }
      C[I*N+J] = c;
}
```

Results – shared memory – C1060

- N=512, double precision
- Different thread geometries
- Baseline: 23 GFlops on 4 cores of Lilliput
  69 Gflops on 8 cores of Triton (double)

Geometry	**16 × 16**	**8 × 8**	**4 × 4**
Uncoalesced	9.2	8.9	8.2
Coalesced	125 (57)	53 (41)	12 (15)



fotosearch.com

Occupancy Calculator

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

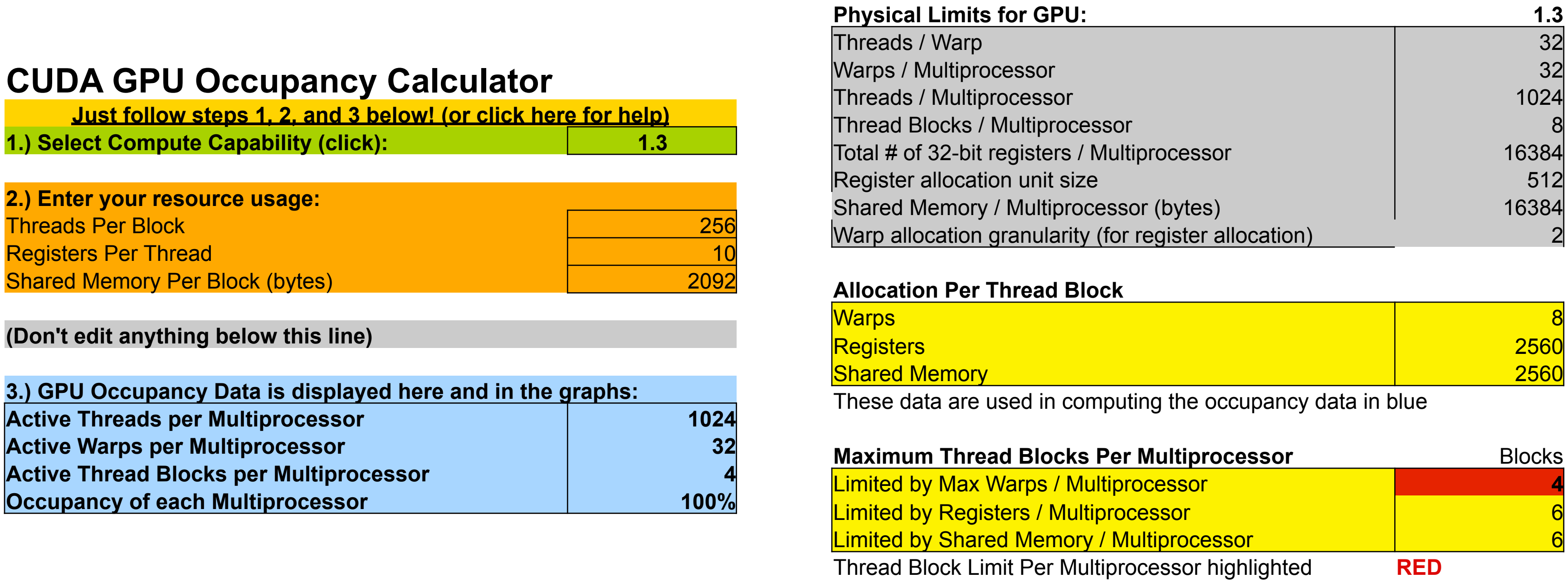
Determining occupancy

- Recall the definition for occupancy
  # active warps ÷ max # warps supported by vector unit
- Maximizing the occupancy doesn't always maximize performance
- NVIDIA provides an occupancy calculator
- Determine resource usage from nvcc

```
nvcc  --ptxas-options=-v
Used 10  registers, 2092+16 bytes smem
```

Occupancy calculation with 16 x 16 threads

$$\text{Occupancy} = \frac{\#\ \text{active warps per SM}}{\text{Maximum possible}\ \#\ \text{active warps}}$$



CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	**1.3**

2.) Enter your resource usage:

Threads Per Block	256
Registers Per Thread	10
Shared Memory Per Block (bytes)	2092

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor	**1024**
Active Warps per Multiprocessor	**32**
Active Thread Blocks per Multiprocessor	**4**
Occupancy of each Multiprocessor	**100%**

Physical Limits for GPU:	**1.3**
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

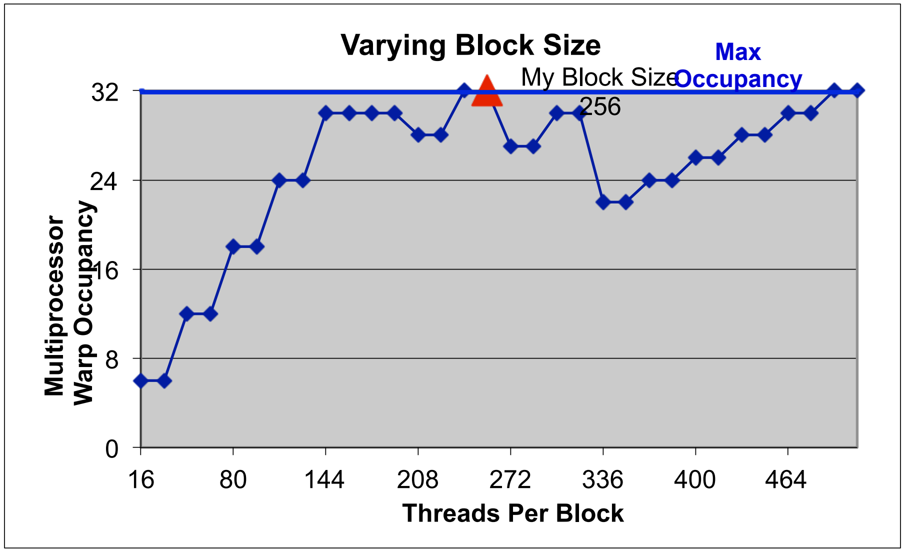
Allocation Per Thread Block	
Warps	8
Registers	2560
Shared Memory	2560

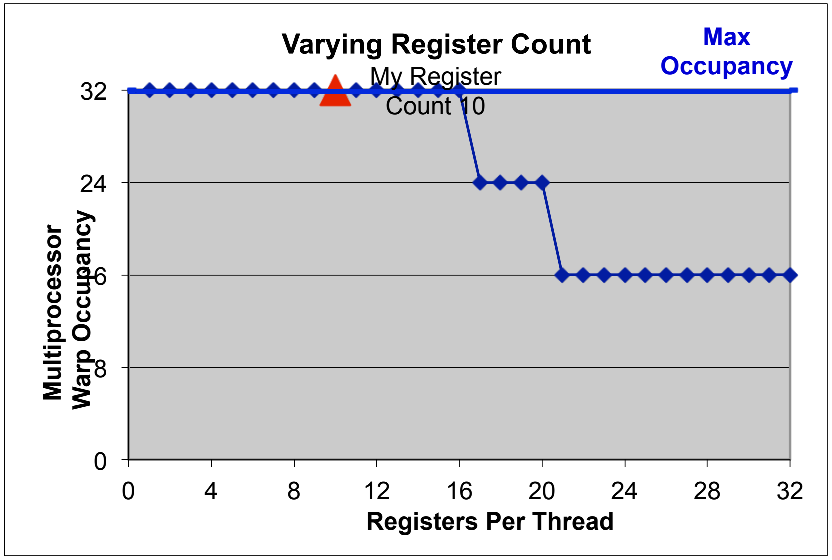
These data are used in computing the occupancy data in blue

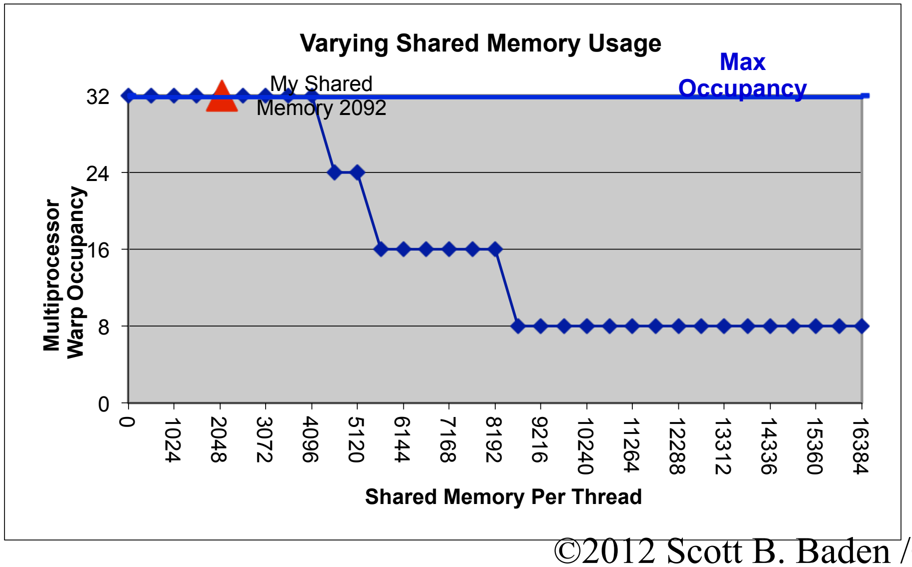
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	6
Limited by Shared Memory / Multiprocessor	6

Thread Block Limit Per Multiprocessor highlighted **RED**

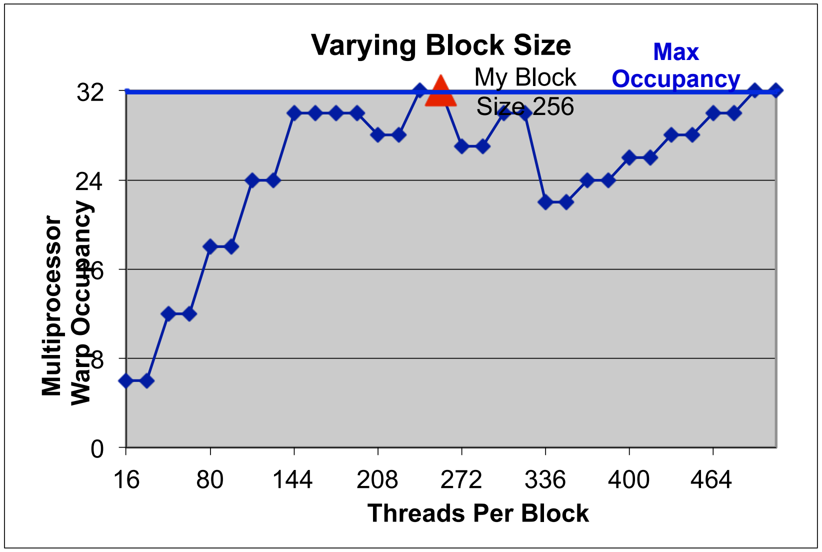
Full occupancy







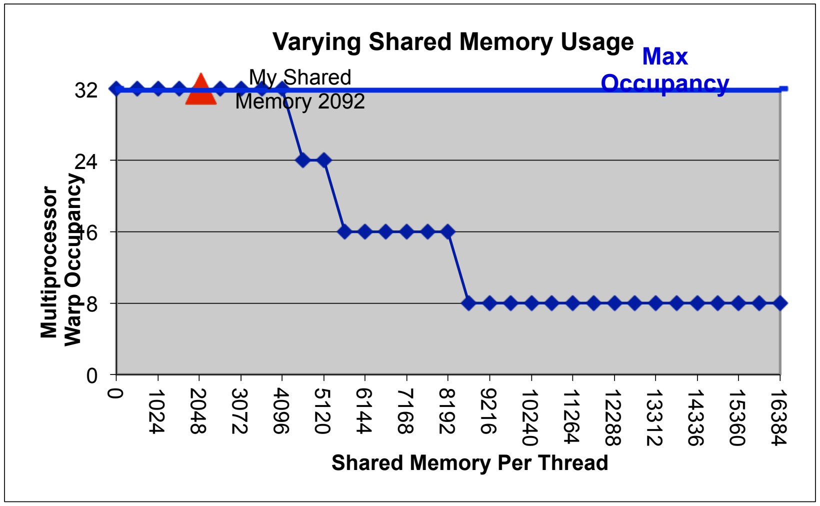
8 x 8 thread blocks

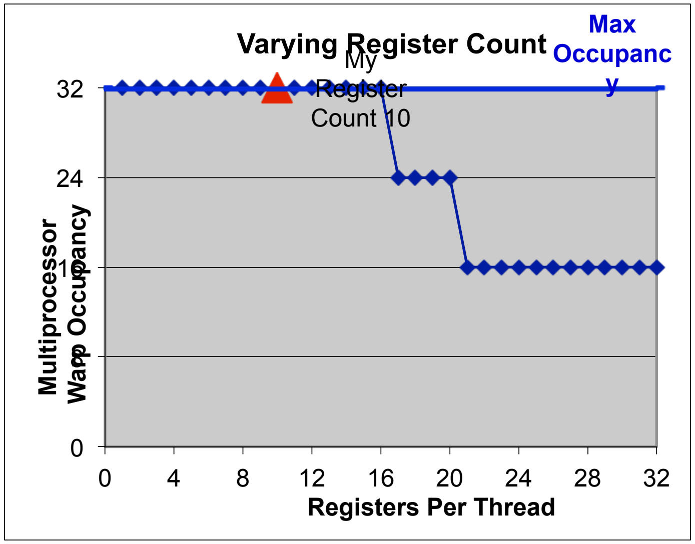


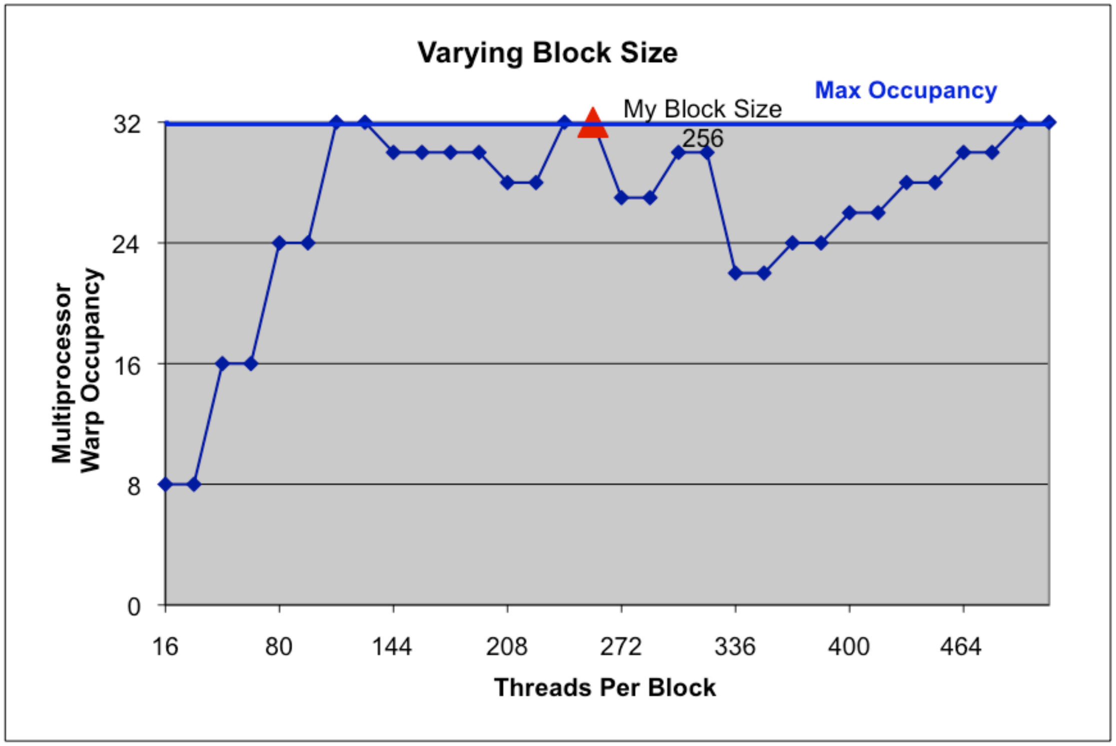
2.) Enter your resource usage:	
Threads Per Block	64
Registers Per Thread	10
Shared Memory Per Block (bytes)	2092

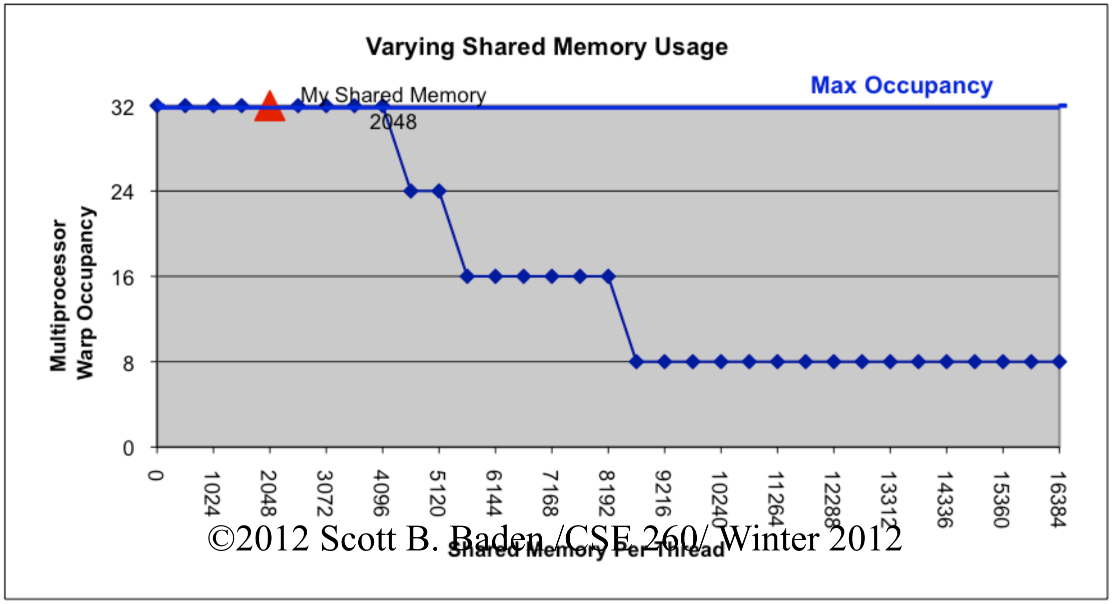
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	8
Limited by Registers / Multiprocessor	16
Limited by Shared Memory / Multiprocessor	**6**

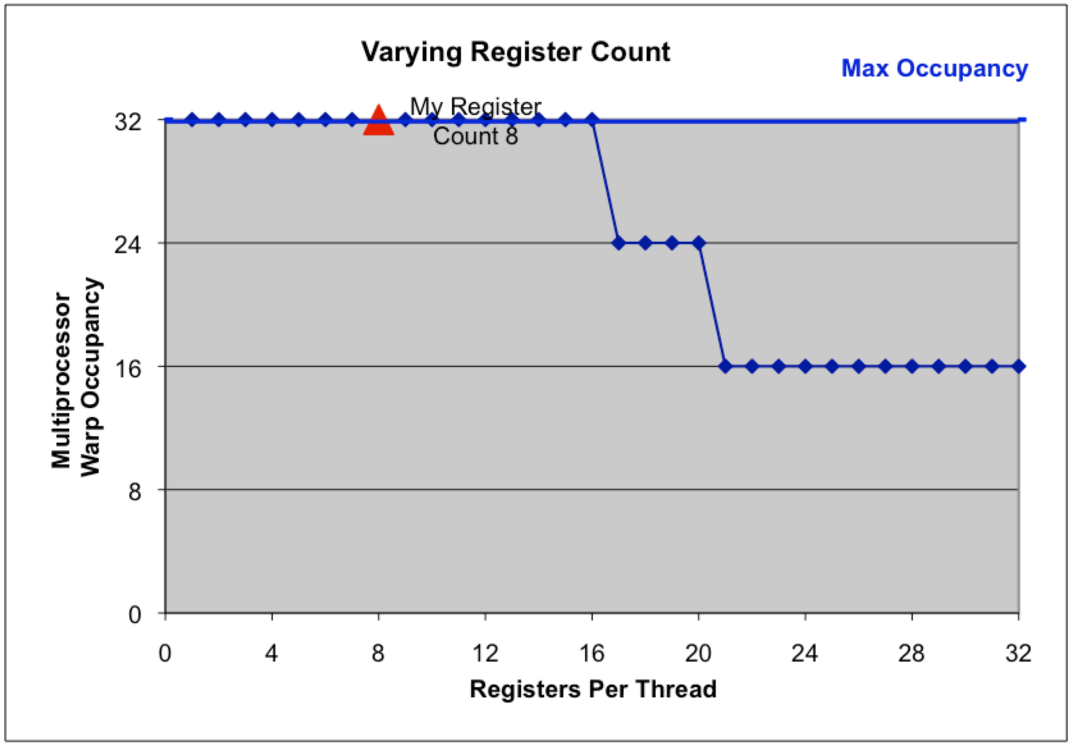
Thread Block Limit Per Multiprocessor highlighted **RED**






Varying Block Size
Max Occupancy
My Block Size
256
Multiprocessor Warp Occupancy
32
24
16
8
0
16
80
144
208
272
336
400
464
Threads Per Block


Varying Shared Memory Usage
Max Occupancy
My Shared Memory
2048
Multiprocessor Warp Occupancy
32
24
16
8
0
0
1024
2048
3072
4096
5120
6144
7168
8192
9216
10240
11264
12288
13312
14336
15360
16384
Shared Memory Per Thread


Varying Register Count
Max Occupancy
My Register Count 8
Multiprocessor Warp Occupancy
32
24
16
8
0
0
4
8
12
16
20
24
28
32
Registers Per Thread