

Benchmarking GPUs to tune dense linear algebra

Yajaira Gonzalez Gonzalez

04/28/2011

Outline

- GPU Architecture
- Strip Mining
- Register File vs Shared Memory
- Global Barrier Synchronization
- Block Matrix Multiplication

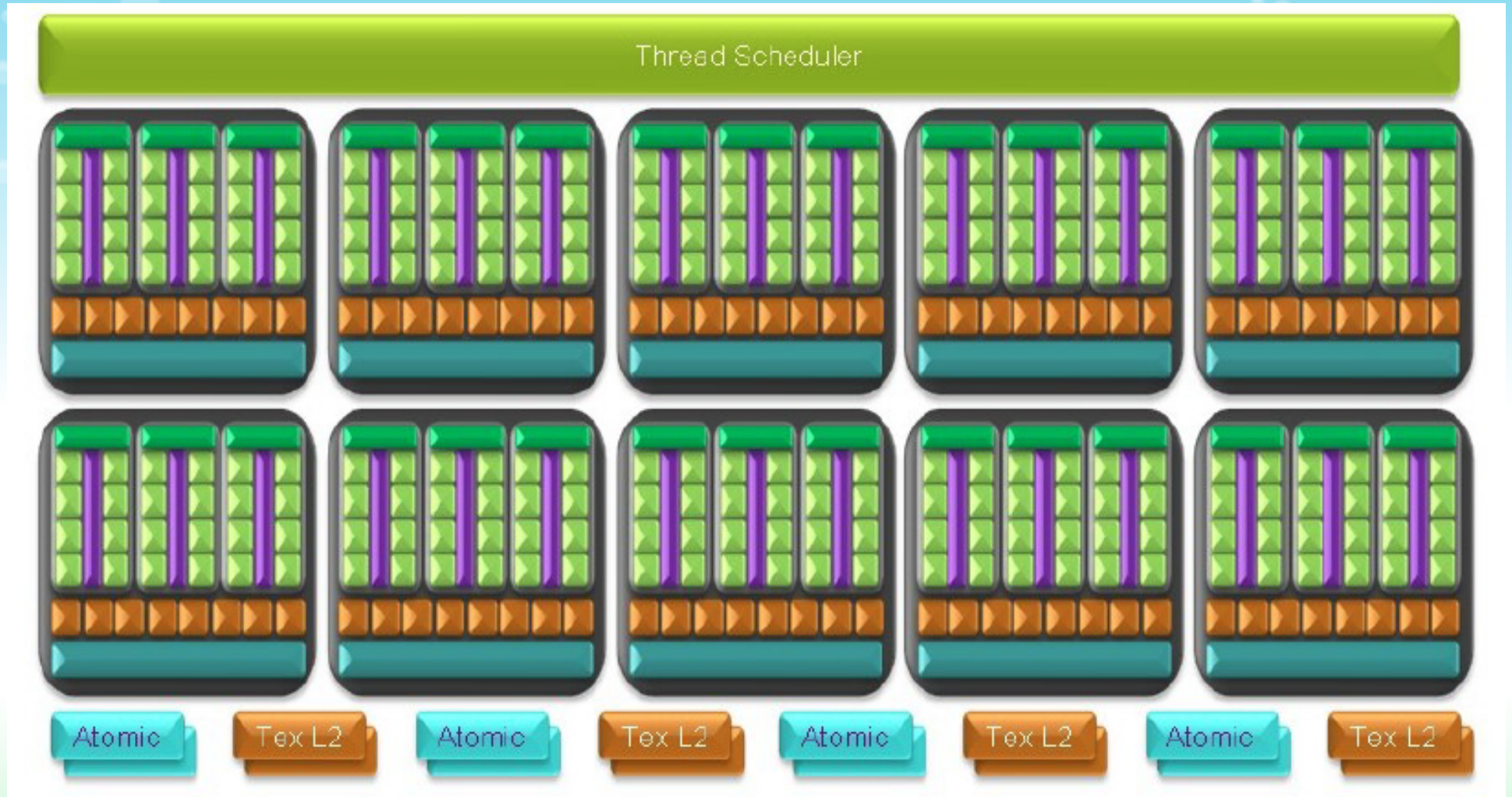
GPUs Used

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.

- 800 series, 900 series and 200 series

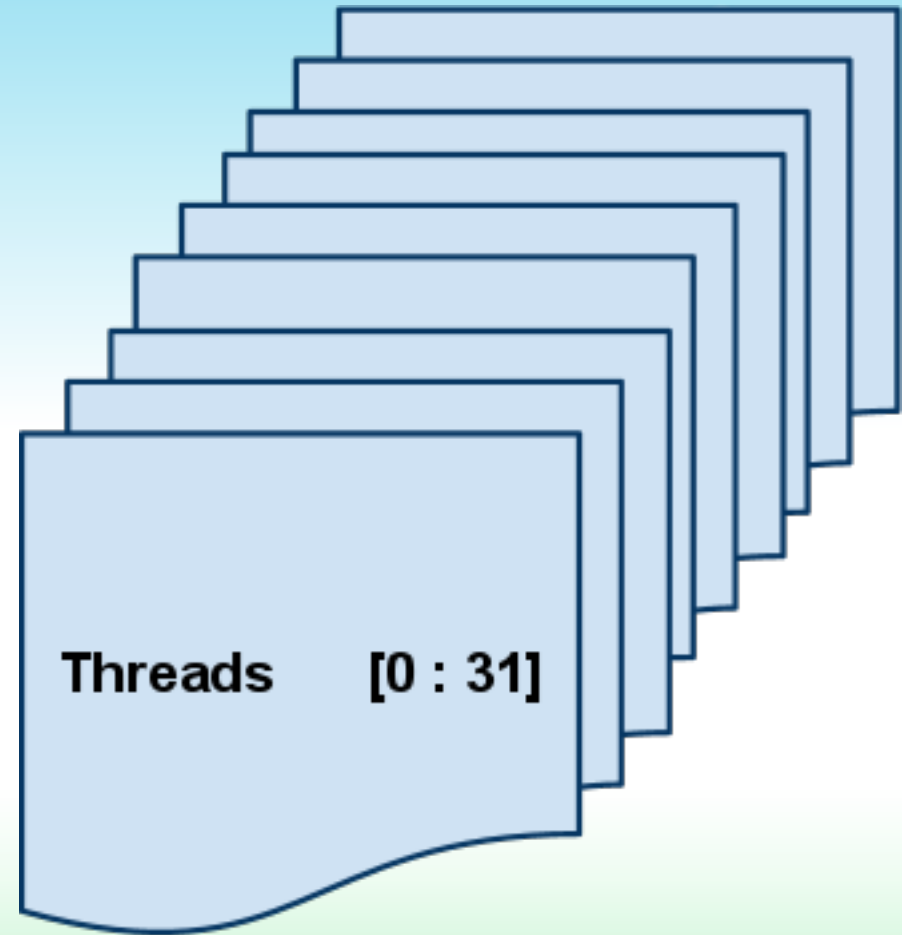
GPU Architecture Review (GTX 280)



TPC's (10) -> SM's (3) -> SP (8)

WARP Review

- Collection of up to 32 threads of the same block.
- Executes one common instruction at a time.



WARP Review

**All threads execute
"int idx = threadIdx.x"**

Thread Code:

```
int idx = threadIdx.x;  
a[idx] = 4 * a[idx];
```

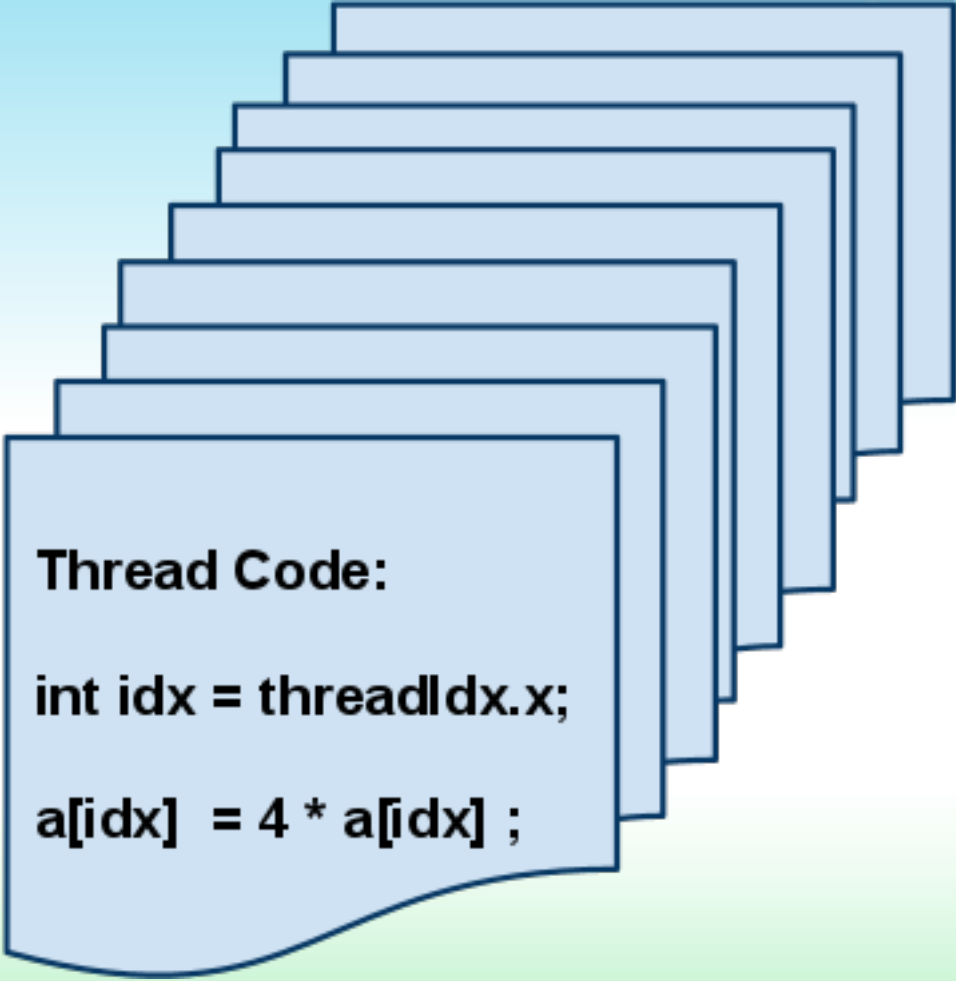
WARP Review

**All threads execute
"a[idx] = 4 * a[idx]"**



Thread Code:

```
int idx = threadIdx.x;  
a[idx] = 4 * a[idx];
```



Strip Mining Idea

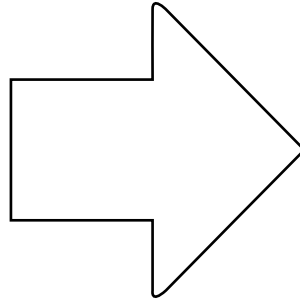


- Use short vector elements to avoid the costs associated with spreading the data across many warps.
- Reduce the number of warps by reducing the number of threads within a block.

Strip Mining

- Basically Unrrolling
 - Reduce the number of iterations to performed
 - Increase work per iteration

```
for( int i = 0; i < n; i++ )  
{  
    y[i] += a*x[i];  
}
```



```
for( int i = 0; i < n; i+=4 )  
{  
    y[i+0] += a*x[i+0];  
    y[i+1] += a*x[i+1];  
    y[i+2] += a*x[i+2];  
    y[i+3] += a*x[i+3];  
}
```

n
threads needed

n / 4
threads needed

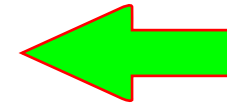
Strip Mining

- Advantages
 - Uses fewer threads.
 - Uses less registers.
 - More concurrent thread blocks can be run.
- Smallest block size that gives good performance:
 - Single Precision: Vector Length = 64 elements
 - Double Precision: Vector Length = 16 elements

Register File vs Shared Memory

- Favors the use of Register Files instead of Shared Memory
 - Reg File: Faster and Larger
 - Reg File : 32 - 64 KB per SM
 - SMEM : 16KB per SM

Operation	8800GTX	other GPUs in Table 1
$a+b*s[i]$	66%	66%
$a+a*s[i]$	66%	75%
$a+s[i]$	74%	99%



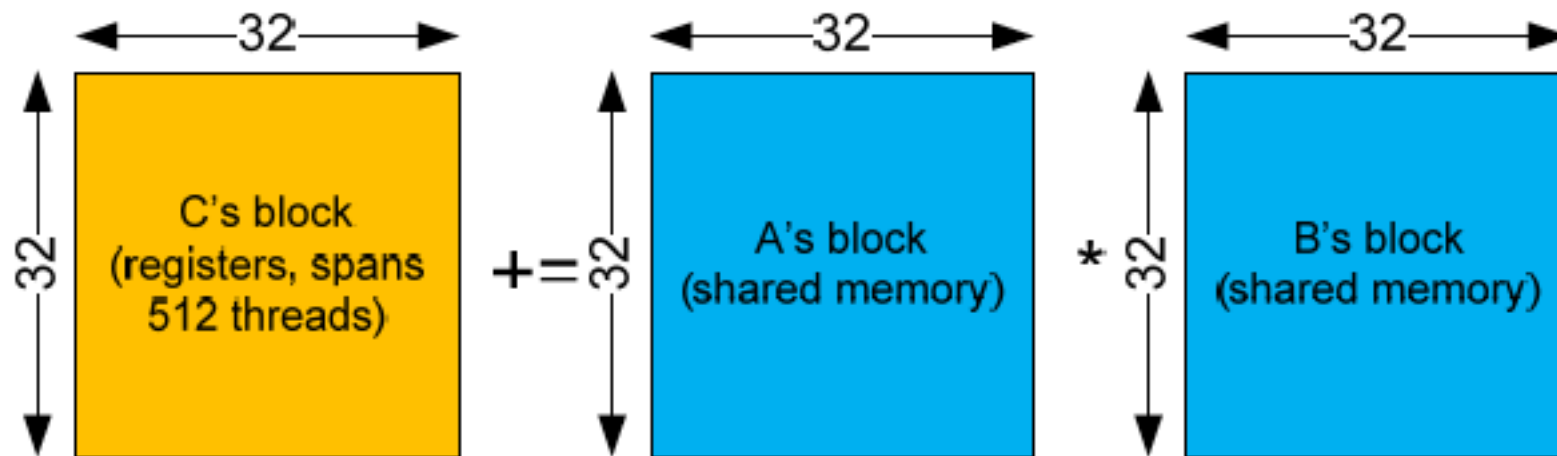
- Newer GPUs handle shared memory faster.
- However, generic MAD achieved 66% of peak.

Global Barrier for Synchronization

- CUDA programming guide suggest the use of kernel launch for synchronization.
- Kernel Launch Overhead
 - Synchronizing at the end: 3 - 7 μ s
 - Synchronizing every time : 10 - 14 μ s
- Global Barrier on GPU implementation:
 - 1.5x - 5.4x less than the cost of a new kernel launch.

Block Matrix Multiplication Routine

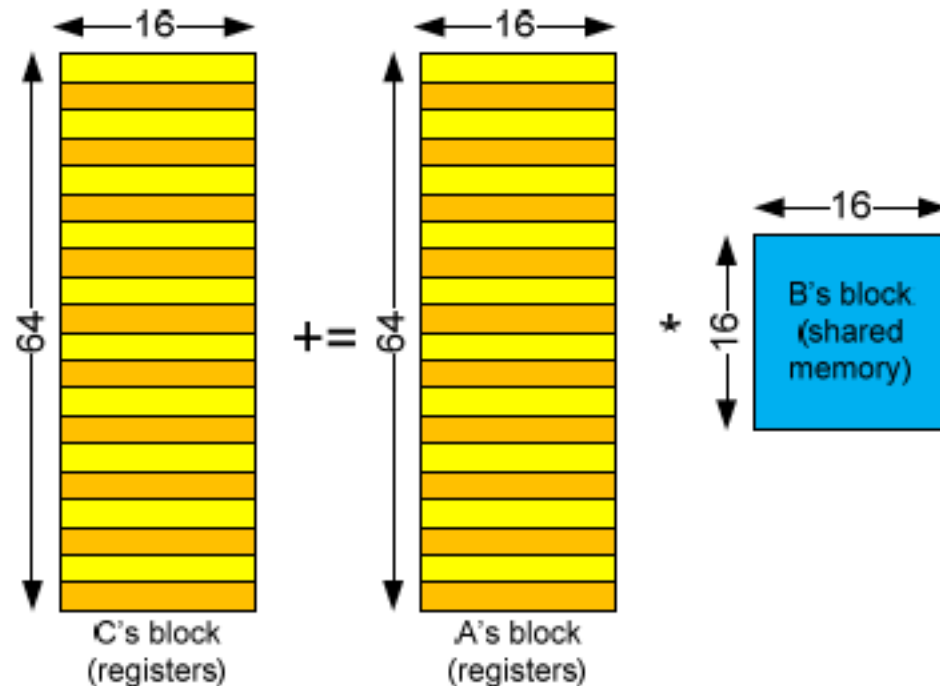
SGEMM in CUBLAS 1.1



- Problems:
 - Two 32x32 blocks consume half of shared memory
 - Only one thread block per multiprocessor at a time
 - 3 reads from shared memory per 2 multiply-and-adds
 - Bound by shared memory access

Block Matrix Multiplication Routine

Our SGEMM



- Blocks in A and C are row-cyclic distributed across threads
 - Don't need to pass A's block via shared memory
- Low usage of shared memory
- Can run many concurrent thread blocks

Results

GPU	SP peak, Gflop/s	SGEMM("N", "N", ...)			SSYRK("L", "N", ...)		DP peak, Gflop/s	DGEMM	DSYRK	
		CUBLAS1.1	ours	estimate	CUBLAS2.0	ours		ours	CUBLAS2.0	ours
8600GTS	93	37%	60%	58%	36%	60%	—	—	—	—
8800GTX	346	37%	60%	58%	37%	60%	—	—	—	—
9800GTX	429	36%	58%	58%	36%	58%	—	—	—	—
GTX280	624	44%	60%	58%	45%	60%	78	97%	35%	95%

Table 2: The estimated and the best observed rates in matrix-matrix multiply routines shown as a fraction of the peak.

- Best performance over different GPUs: ~58%-60 %.
- Performance scales linearly with the number of core and clock rate.

Results

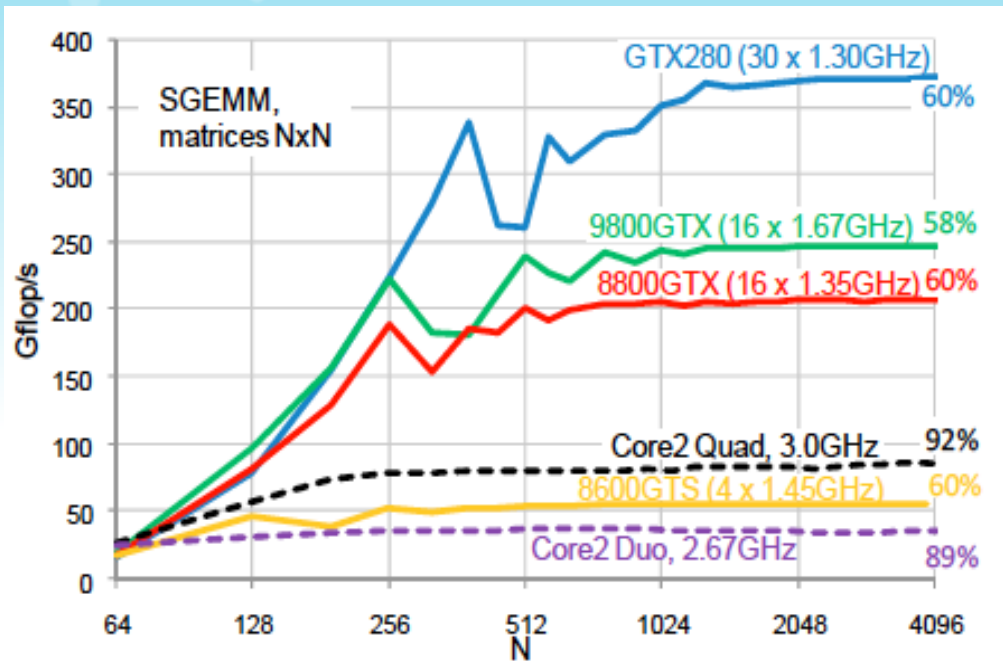


Figure 5: Rates in single precision matrix-matrix multiply on GPUs and CPUs. Percents indicate the fractions of peak.

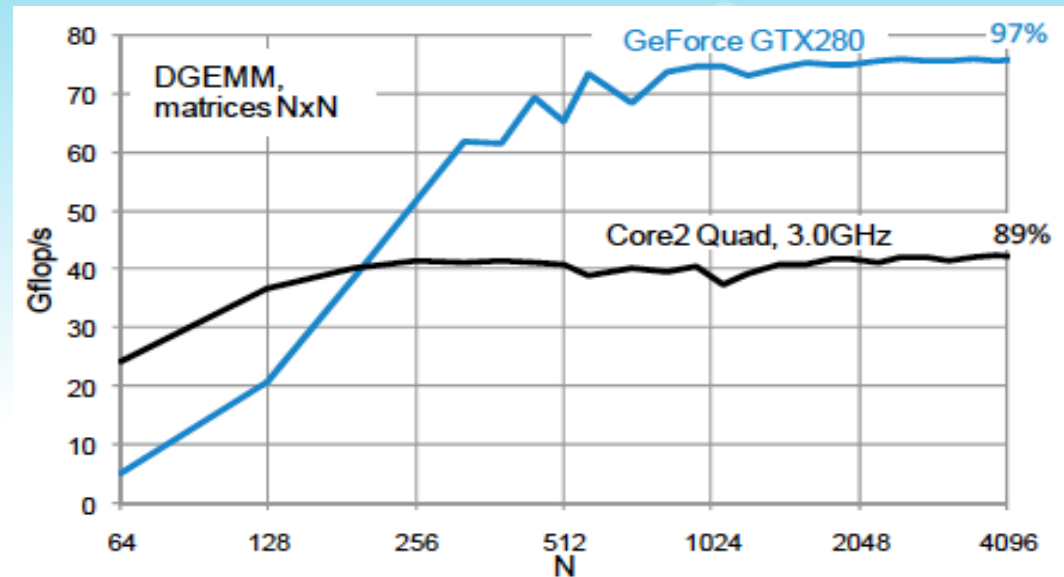


Figure 6: Rates in double precision matrix-matrix multiply on a GPU and a CPU. Percents indicate the fractions of peak.

- Absolute Rates achieved.
- Performance Scales linearly with the number of cores and clock rate.

The top corners of the slide are decorated with clusters of light blue and white bubbles of varying sizes, creating a soft, aquatic feel.

Thank you!

Additional

```
Vector length: 64 //stripmined into two warps by GPU
Registers: a, c[1:16] //each is 64-element vector
Shared memory: b[16][16] //may include padding

Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
  b[1:16][1:16] = next 16×16 block in B or BT
  local barrier //wait until b[][] is written by all warps
  unroll for i = 1 to 16 do
    a = next 64×1 column of A
    c[1] += a*b[i][1] // rank-1 update of C's block
    c[2] += a*b[i][2] // data parallelism = 1024
    c[3] += a*b[i][3] // stripmined in software
    ... // into 16 operations
    c[16] += a*b[i][16] // access to b[][] is stride-1
  endfor
  local barrier //wait until done using b[][]
  update pointers in A and B
repeat until pointer in B is out of range
Merge c[1:16] with 64×16 block of C in memory
```

Figure 4: The structure of our matrix-matrix multiply routines.

Additional

	CUBLAS 1.1	Our code
C 's block, where stored	32×32 , regs	64×16 , regs
A 's block, where stored	32×32 , smem	64×1 , regs
B 's block, where stored	32×32 , smem	16×16 , smem
Vector length	512	64
Scalar registers per scalar thread	15	30
Registers per vector thread	30 KB	7.5 KB
smem per vector thread	8.3 KB	1.1 KB
Threads/core, 8800GTX	1	4
Warps/core, 8800GTX	16	8
Instructions in inner loop	115	312
MAD instructions	64	256
smem-to-register MOVs	32	0
Expected, % of peak	44%	58%
Sustained, % of peak	36–44%	58–60%

Table 3: Details of our code and the code in CUBLAS 1.1. Instructions counts are for the inner loop only and were obtained using *decuda*. A 's 64×1 blocks are given as defined in the C-level program. This block size is increased when compiling by unrolling the loop and assigning the blocks fetched in different iterations to different registers.

GPU Memory System

- Larger latencies, more cache misses.
- Array Sizes reveals cache sizes. (5KB and 192 KB)

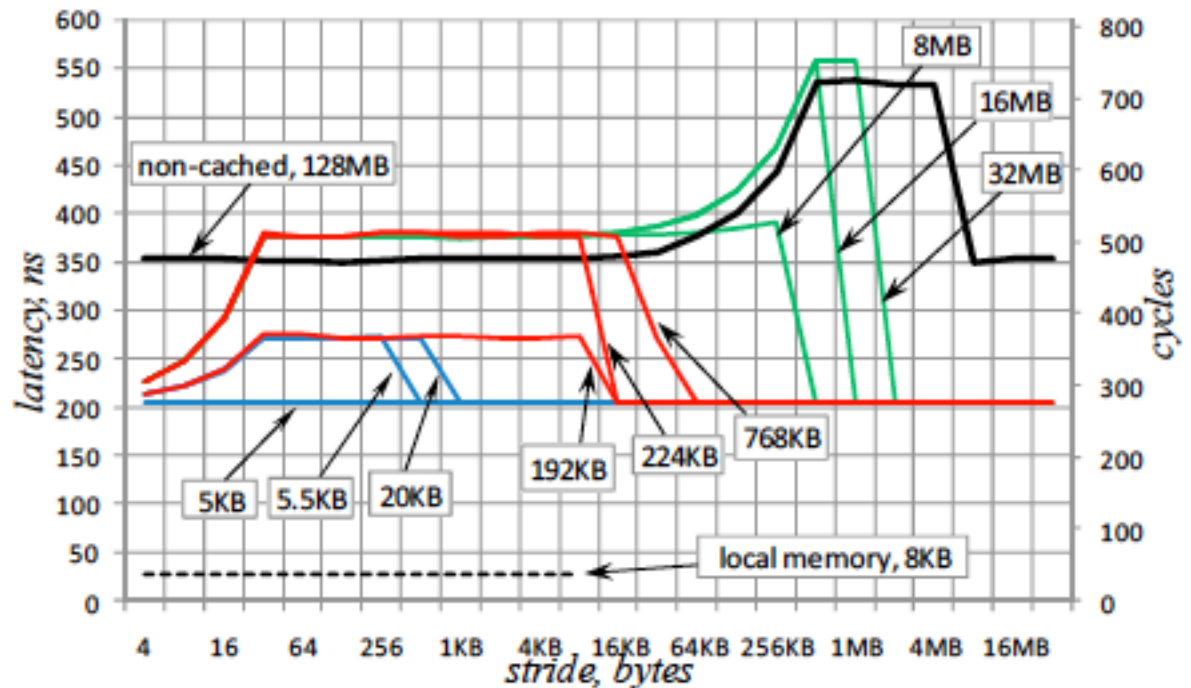


Figure 1: Memory latency as revealed by the pointer chasing benchmark on GeForce 8800 GTX for different kinds of memory accesses. Array size is shown in the boxes. Cached access assumed unless otherwise specified. Blue, red and green lines highlight 5KB cache, 192 KB cache, and 512KB memory pages respectively. Solid black is non-cached access, dashed black is local memory.