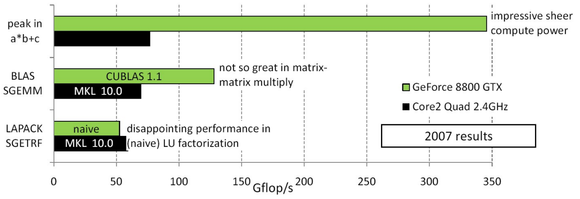
SGEMM

- Vasily Volkov, James Demmel: *Benchmarking GPUs to tune dense linear algebra*. SC 2008

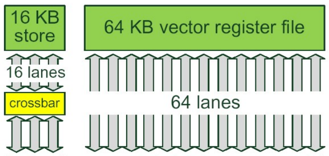
Some following slides are from the presentation of the above paper at SC'08.



- Memory bandwidth scales with number of memory controllers

GPU (NVIDIA GeForce)	8600 GTS	9800 GTX	GTX 280
Processor cores	4	16	30
Compute capability (a+b*c)	93 Gflop/s	429 Gflop/s	624 Gflop/s
Memory controllers	2	4	8
Memory bandwidth	32 GB/s	70 GB/s	141 GB/s

GPU Memory Hierarchy



- Register file is the fastest and the largest on-chip memory
  - **Keep as much data as possible in registers**
  - However, register file is constrained to vector operations
  - Can live with it — vectorized codes are common in HPC
- Shared memory permits indexed and shared access
  - However, it is 2–4x smaller and has 4x lower bandwidth than registers
    - Only 1 operand in shared memory is allowed versus 4 register operands
  - Moreover, some instructions run slower if using shared memory
  - **Use shared memory as a communication device only**
  - Avoid communication to improve performance

Peak Throughput in Multiply-and-Add

- How much parallelism is *enough* to get the peak?
- Run **1 thread per processor core**
  - Purpose: smallest amount that can control all computing resources
- Assume **sufficient instruction-level parallelism** in the program
  - Purpose: hide pipeline latency
- Choose the shortest vector length that yields the peak
  - Purpose: satisfy inherent data-parallelism constraints
- Result: **98% of arithmetic peak at VL = 64**
  - Therefore, VL=64 is recommended for all compute-bound codes
- However, we never could surpass 66% of peak is using an operand in shared memory
  - We believe this is an inherent bottleneck in the architecture
  - We use this number in the throughput bounds below

Matrix-Matrix Multiply: $C = C + A*B$

- GPU requires using block algorithms in matrix-matrix multiply:
  - Peak rates on one of the latest GPUs are 624 Gflop/s and 141 GB/s
  - This corresponds to 0.23 bytes per flop
  - But naïve matrix-matrix multiply requires 4 bytes per flop
  - Thus, it is bandwidth-bound unless data is reused 18 times
  - Using *MxN* blocks in *C* yields 2/(1/*M*+1/*N*) average reuse
- Use vector algorithms to efficiently use vector registers
  - Such as used on IBM 3090 Vector Facility and Cray X1:
  - Keep *A*'s and *C*'s blocks in registers
  - Keep *B*'s block in a shared storage
  - No other sharing is needed if *C*'s height = VL. We know VL=64 is best
- Choose large enough width of *C*'s block
  - 16 is enough as 2/(1/64+1/16) = 26-way reuse
- Choose a convenient thickness for *A*'s and *B*'s blocks

```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
      A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
      B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
      C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
      __shared__ float bs[16][17];
      float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
      const float *Blast = B + k;
      do
      {
#pragma unroll
            for( int i = 0; i < 16; i += 4 )
                  bs[threadIdx.x][threadIdx.y+i]  = B[i*ldb];
            B += 16;
            __syncthreads();
#pragma unroll
            for( int i = 0; i < 16; i++, A += lda )
            {
                  c[0] += A[0]*bs[i][0];    c[1] += A[0]*bs[i][1];    c[2] += A[0]*bs[i][2];    c[3] += A[0]*bs[i][3];
                  c[4] += A[0]*bs[i][4];    c[5] += A[0]*bs[i][5];    c[6] += A[0]*bs[i][6];    c[7] += A[0]*bs[i][7];
                  c[8] += A[0]*bs[i][8];    c[9] += A[0]*bs[i][9];    c[10] += A[0]*bs[i][10];c[11] += A[0]*bs[i][11];
                  c[12] += A[0]*bs[i][12];c[13] += A[0]*bs[i][13];c[14] += A[0]*bs[i][14];c[15] += A[0]*bs[i][15];
            }
            __syncthreads();
      } while( B < Blast );
      for( int i = 0; i < 16; i++, C += ldc )
            C[0] = alpha*c[i] + beta*C[0];
}
```

Compute pointers to the data

Declare the on-chip storage

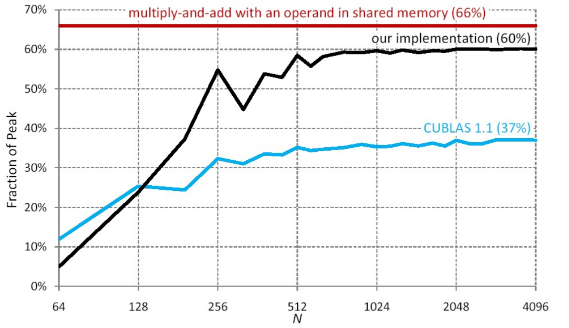
Read next B's block

The bottleneck:
Read A's columns
Do Rank-1 updates

Store C's block to memory

Our code vs. CUBLAS 1.1

Performance in multiplying two *N*x*N* matrices on GeForce 8800 GTX:

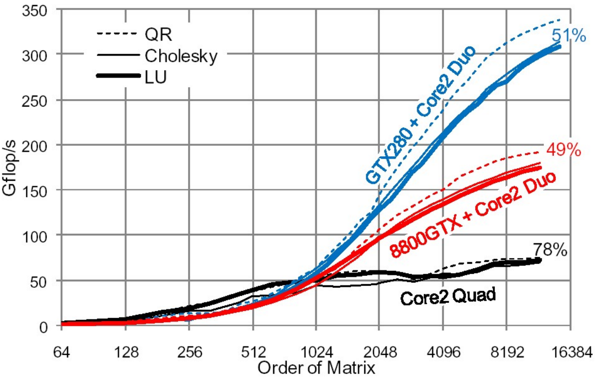


What causes CUBLAS 1.1 to run slower than our code?

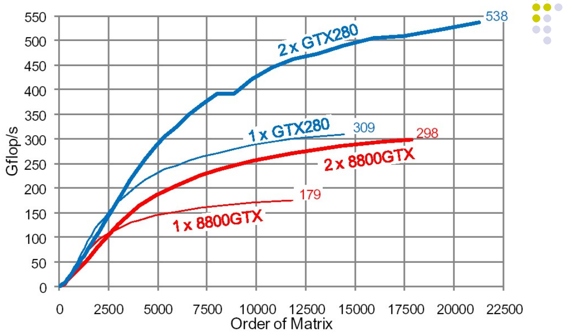
Fast Matrix Factorizations using GPUs

- Use GPU to compute matrix-matrix multiplies only
- Factorize panels on the CPU
- Use look-ahead to overlap computations on CPU and GPU
- Use right-looking algorithms to have more threads in SGEMM
  - Better load balance in the GPU workload, better latency hiding
- Use row-major layout on GPU in LU factorization
  - Requires extra (but fast) matrix transpose for each CPU-GPU transfer
- Substitute triangular solves $LX=B$ with multiply by L^{-1}
  - Provably stable if we do this only when $||L^{-1}|| < fixed\_threshold$
  - Small pivot growth nearly always assumes small $||L^{-1}||$
  - Accuracy of LU assumes small pivot growth anyway
- Use two-level and variable size blocking as finer tuning
  - Thicker blocks impose lower bandwidth requirements in SGEMM
  - Variable size blocking improves CPU/GPU load balance
- Use column-cyclic layout when computing using two GPUs
  - Requires no data exchange between GPUs in pivoting
  - Cyclic layout is used on GPUs only so does not affect panel factorizations

Performance Results



LU Factorization using Two GPUs



- Second GPU allows 1.7x higher rates
- More than half-teraflop using two GPUs

Advanced Tips

- Every round reading 2KB with 256 threads grouped into 2 blocks, each of 128 threads
- At least 2 blocks if using blockwise synchronization
- Total number of active threads >192 for pipeline efficiency

X. Cui, Y. Chen and H. Mei. Improving performance of matrix multiplication and FFT on GPU.

15th ICPADS, pp 42-48, IEEE Computer Society, 2009.

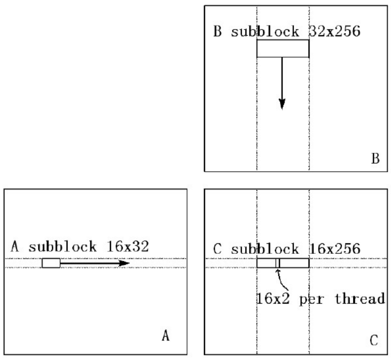
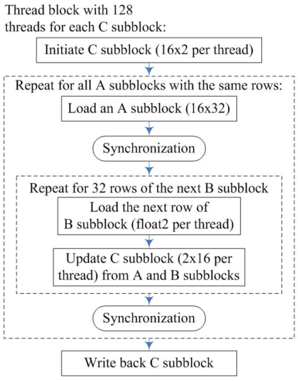


Figure 3: Matrix multiplication: A×B=C with all arrays in row major



Thread block with 128 threads for each C subblock:

Initiate C subblock (16x2 per thread)

Repeat for all A subblocks with the same rows:

Load an A subblock (16x32)

Synchronization

Repeat for 32 rows of the next B subblock

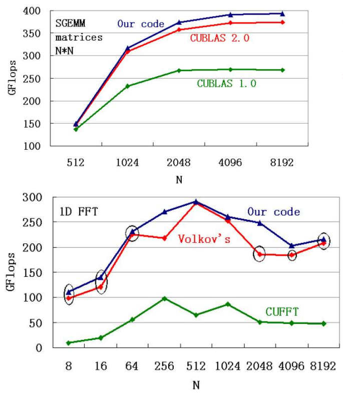
Load the next row of B subblock (float2 per thread)

Update C subblock (2x16 per thread) from A and B subblocks

Synchronization

Write back C subblock

Figure 4: Thread block function for matrix-matrix multiplication.



On GTX 285

SGEMM

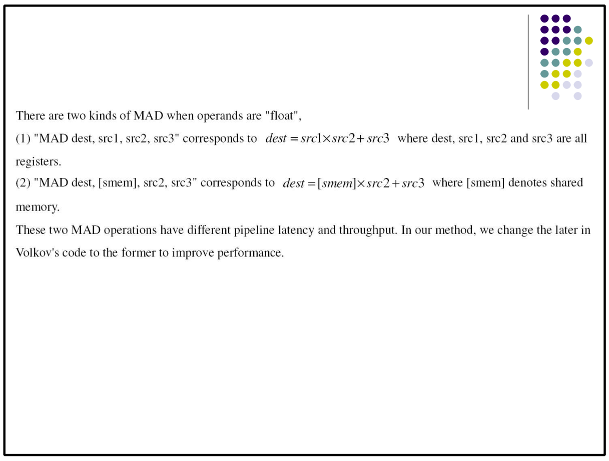
= 448 Gflops!

Hand-Tuned SGEMM on GT200 GPU

Lung-Sheng Chien

Department of Mathematics, Tsing Hua university, R.O.C. (Taiwan)

	GTX295[1]	GTX285	TeslaC1060
# of Streaming Processor	240	240	240
Core Frequency	1242MHz	1476 MHz	1.3 GHz
Memory Speed	999MHz	1242 MHz	800 MHz
Memory Interface	448-bit (7 channel)	512-bit (8 channel)	512-bit (8 channel)
Memory Bandwidth (GB/s)	112	159	102
SP, peak (Gflop/s)	894	1063	933
SP without dual issue	596.2	708.5	624
DP, peak (Gflop/s)	74.5	88.6	78
DRAM (MByte)	896	1024	4096

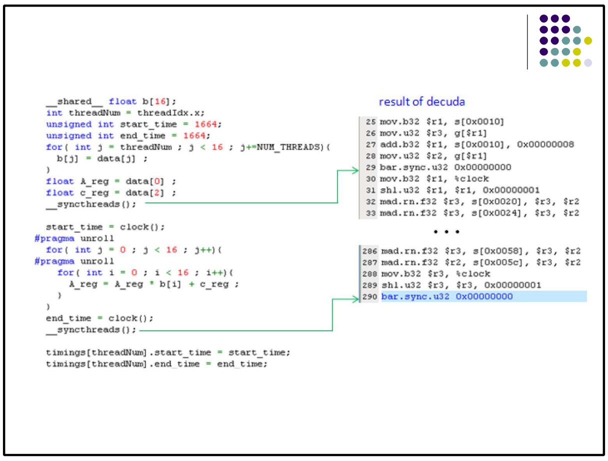


There are two kinds of MAD when operands are "float",

(1) "MAD dest, src1, src2, src3" corresponds to $dest = src1 \times src2 + src3$ where dest, src1, src2 and src3 are all registers.

(2) "MAD dest, [smem], src2, src3" corresponds to $dest = [smem] \times src2 + src3$ where [smem] denotes shared memory.

These two MAD operations have different pipeline latency and throughput. In our method, we change the later in Volkov's code to the former to improve performance.



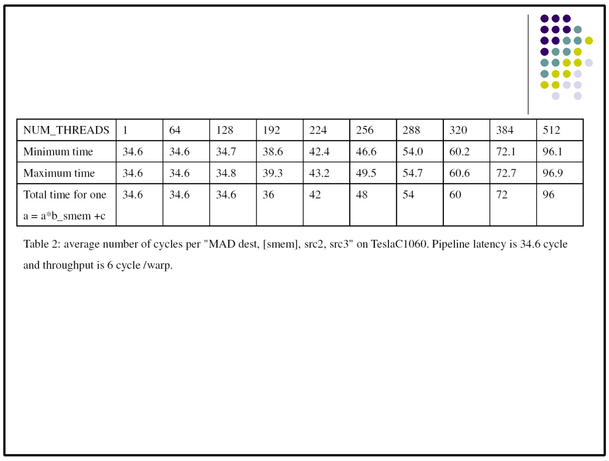
```
  __shared__ float b[16];
  int threadNum = threadIdx.x;
  unsigned int start_time = 1664;
  unsigned int end_time = 1664;
  for( int j = threadNum ; j < 16 ; j+=NUM_THREADS){
    b[j] = data[j] ;
  }
  float A_reg = data[0] ;
  float c_reg = data[2] ;
  __syncthreads();

  start_time = clock();
#pragma unroll
  for( int j = 0 ; j < 16 ; j++){
#pragma unroll
    for( int i = 0 ; i < 16 ; i++){
      A_reg = A_reg * b[i] + c_reg ;
    }
  }
  end_time = clock();
  __syncthreads();

  timings[threadNum].start_time = start_time;
  timings[threadNum].end_time = end_time;
```

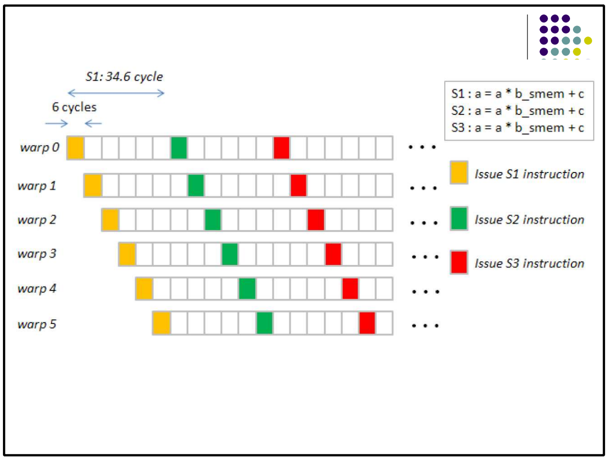
result of decuda

```
25 mov.b32 $r1, s[0x0010]
26 mov.u32 $r3, g[$r1]
27 add.b32 $r1, s[0x0010], 0x00000008
28 mov.u32 $r2, g[$r1]
29 bar.sync.u32 0x00000000
30 mov.b32 $r1, %clock
31 shl.u32 $r1, $r1, 0x00000001
32 mad.rn.f32 $r3, s[0x0020], $r3, $r2
33 mad.rn.f32 $r3, s[0x0024], $r3, $r2
              • • •
286 mad.rn.f32 $r3, s[0x0058], $r3, $r2
287 mad.rn.f32 $r2, s[0x005c], $r3, $r2
288 mov.b32 $r3, %clock
289 shl.u32 $r3, $r3, 0x00000001
290 bar.sync.u32 0x00000000
```



NUM_THREADS	1	64	128	192	224	256	288	320	384	512
Minimum time	34.6	34.6	34.7	38.6	42.4	46.6	54.0	60.2	72.1	96.1
Maximum time	34.6	34.6	34.8	39.3	43.2	49.5	54.7	60.6	72.7	96.9
Total time for one a = a*b_smem +c	34.6	34.6	34.6	36	42	48	54	60	72	96

Table 2: average number of cycles per "MAD dest, [smem], src2, src3" on TeslaC1060. Pipeline latency is 34.6 cycle and throughput is 6 cycle /warp.



S1: 34.6 cycle

6 cycles

warp 0

warp 1

warp 2

warp 3

warp 4

warp 5

S1 : a = a * b_smem + c
S2 : a = a * b_smem + c
S3 : a = a * b_smem + c

Issue S1 instruction

Issue S2 instruction

Issue S3 instruction

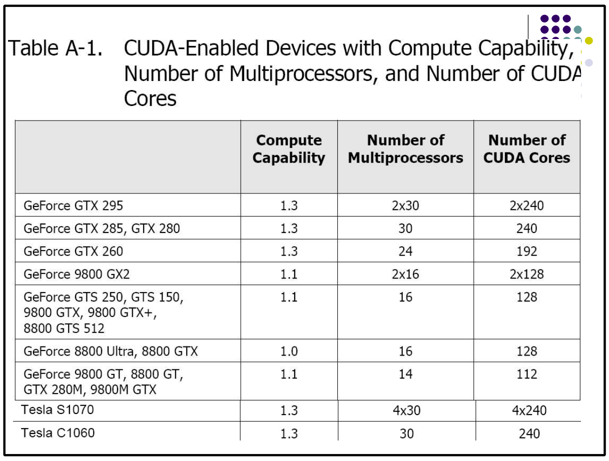
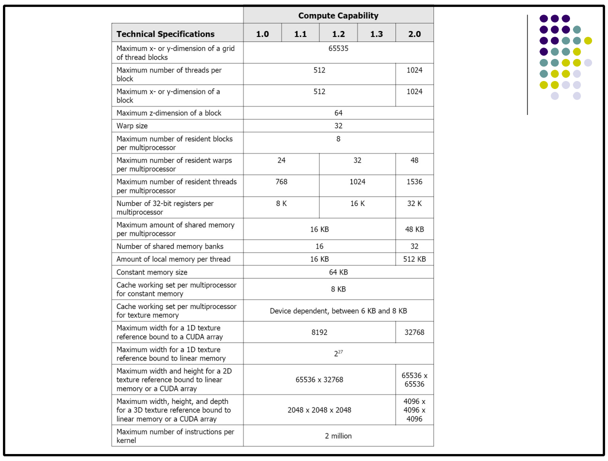
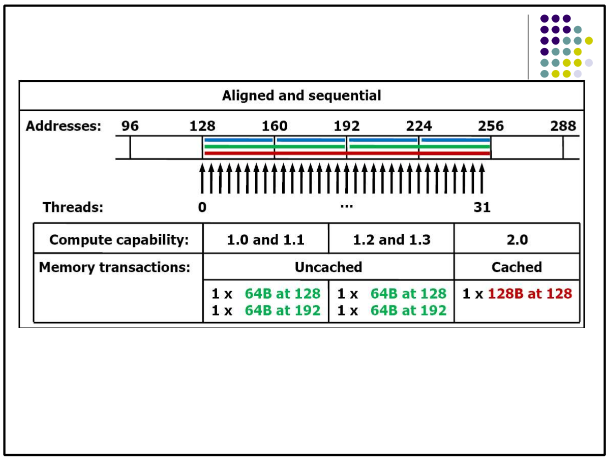


Table A-1. CUDA-Enabled Devices with Compute Capability, Number of Multiprocessors, and Number of CUDA Cores

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	1.1	14	112
Tesla S1070	1.3	4x30	4x240
Tesla C1060	1.3	30	240

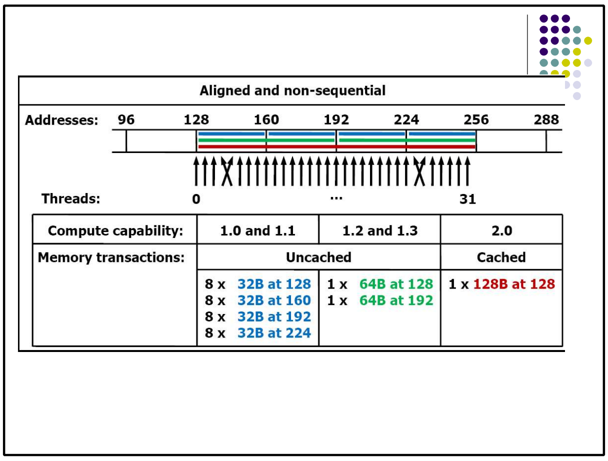


	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of instructions per kernel	2 million				



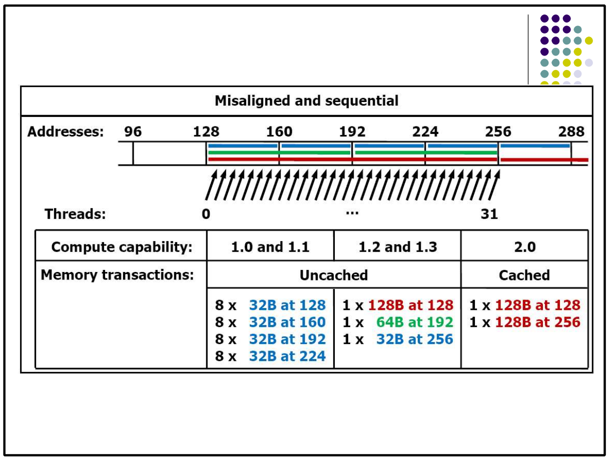
Aligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128



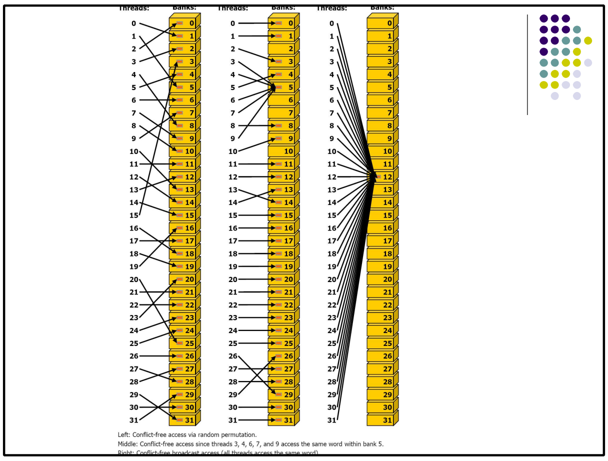
Aligned and non-sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128



Misaligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256



Left: Conflict-free access via random permutation.

Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Right: Conflict-free broadcast access (all threads access the same word).