Lecture 13

Numerical Linear Algebra

Announcements

- Progress report due in class on Thu 11/12
- No class on 11/17 and 11/19
- Makeup lectures
  - Friday 11/20. 3:00 to 4:20
  - Weds 12/2 5-7PM
- CSE 260 Symposium
  - Week 10: Tues, Weds, Thurs

Background on Gaussian Elimination

Linear systems of equations

- A common task in scientific computation is to solve a system of linear equations
- Often result from of discretizing a differential equation
- Example: linear system of 2 equations in 2 unknowns

  (1) $2x + 3y = 8$
  (2) $3x + 2y = 7$

- Rewriting equation (1)

  $x = (8-3y)/2$

- Substituting x into the LHS of equation (2)

  $3(8-3y)/2 + 2y = (24-9y)/2 + 2y$

  $\Rightarrow$ $(24-9y) + 4y = 14 \Rightarrow 10 = 5y \Rightarrow y = 2$

- Back substituting the value of y into equation (1)

  $x = 1$

Matrix vector equations

- Our linear system of 2 equations in 2 unknowns …

  $2x_1 + 3x_2 = 8$

  $3x_1 + 2x_2 = 7$

- may be conveniently expressed in matrix notation: A**x** = b

$$A = \begin{pmatrix} 2 & 3 \\ 3 & 2 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, b = \begin{pmatrix} 8 \\ 7 \end{pmatrix}$$

- When we solved for $x_1 = (8\text{-}3\ x_2)/2$ and substituted into the 2nd equation, we reduced the matrix to an equivalent form

$$A = \begin{pmatrix} 2 & 3 \\ 0 & -2.5 \end{pmatrix}, b = \begin{pmatrix} 8 \\ -5 \end{pmatrix}$$

- We multiplied row 1 of A by 3/2 and subtracting the scaled version from row 2 of **A** and **b**

Rank 1 updates

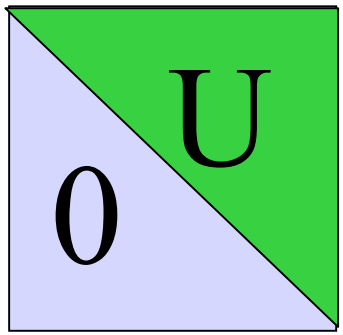
- We call this a ***rank-1 update***
- Multiplying row 1 by 3/2: [3 9/2]
- Subtracting from row 2:
- Similarly for **b**

$$A = \begin{pmatrix} 2 & 3 \\ 3 & 2 \end{pmatrix}$$

$$A' = \begin{pmatrix} 2 & 3 \\ 0 & -2.5 \end{pmatrix}$$

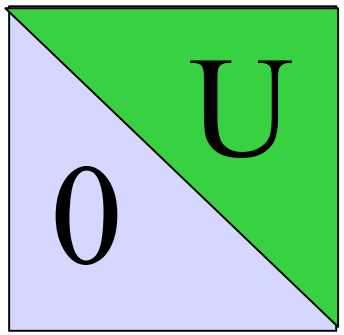
Gaussian Elimination

- The process of eliminating the non-zero values under the main diagonal is called ***Gaussian Elimination***, named after the mathematician
  Johann Carl Friedrich Gauss (1777-1855)
- Input: an n × n matrix corresponding to a linear system of n equations in n unknowns (must have non-trivial sol'n)
- Eliminate the non-zero values under the main diagonal to produce an
  upper triangular matrix **U**



Solving the system of linear equations

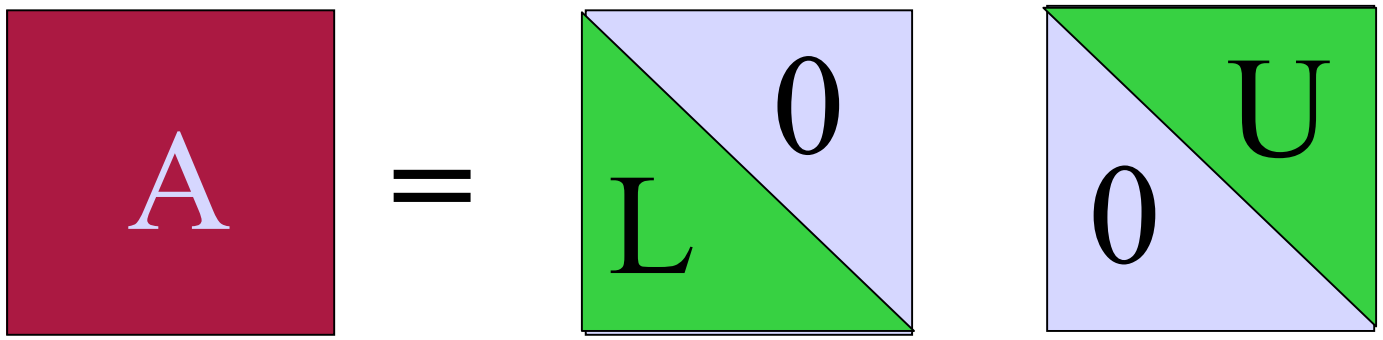
- Step 1: obtain the upper triangular matrix U …
- Step 2: solve the corresponding upper triangular system U**x** = **c** by ***back substitution***



What are we computing?

- GE computes the ***LU factorization*** A = L U, where L is a ***lower triangular matrix***
- Plugging LU into the original equation A**x** = **b**

  A**x** = (LU) **x** = L (U**x**) = L**y** = **b** where **y** = U**x**

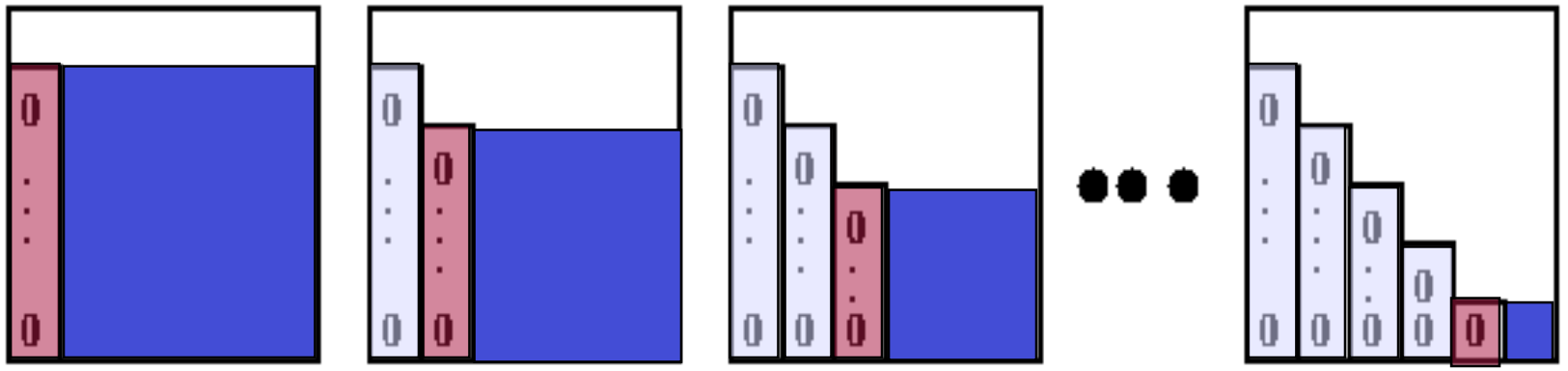


Cost

- To solve A**x** = **b**
  - Factorize A = LU using GE *(2/3 n^3 flops)*
  - Solve L**y** = **b** for **y** using substitution (n^2 flops)
  - Solve U**x** = **y** for **x** using back substitution (n^2 flops)
- We don't compute U explicitly unless we are solving for multiple right hand sides **b**

Visualizing the algorithm

- Eliminating non-zeroes below the diagonal …
  - One column at a time
  - Scanning from left to right



Column 0 Column 1 Column 2 Column n-1

A 3 × 3 example

- Consider the following system of equations

$$
\begin{aligned}
x_0 + x_1 + x_2 &= 3 \\
4x_0 + 3x_1 + 4x_2 &= 8 \\
9x_0 + 3x_1 + 4x_2 &= 7
\end{aligned}
$$

- We usually write the system as an ***augmented matrix***

$$
\left[\begin{array}{ccc|c}
1 & 1 & 1 & 3 \\
4 & 3 & 4 & 8 \\
9 & 3 & 4 & 7
\end{array}\right]
$$

3 × 3 example

- Multiply row 0 by 4, and subtract from row 1

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 4 & 3 & 4 & 8 \\ 9 & 3 & 4 & 7 \end{array}\right]$$

[4 3 4 8] – 4*[1 1 1 3] = [0 -1 0 -4]

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 9 & 3 & 4 & 7 \end{array}\right]$$

3 × 3 example

- Multiply row 0 by 9, and subtract from row 2

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 9 & 3 & 4 & 7 \end{array}\right]$$

[9 3 4 7] – 9*[1 1 1 3] = [0 -6 -5 -20]

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & -6 & -5 & -20 \end{array}\right]$$

3 × 3 example

- Eliminate second column
- Multiply row 1 by 6, and add to row 2

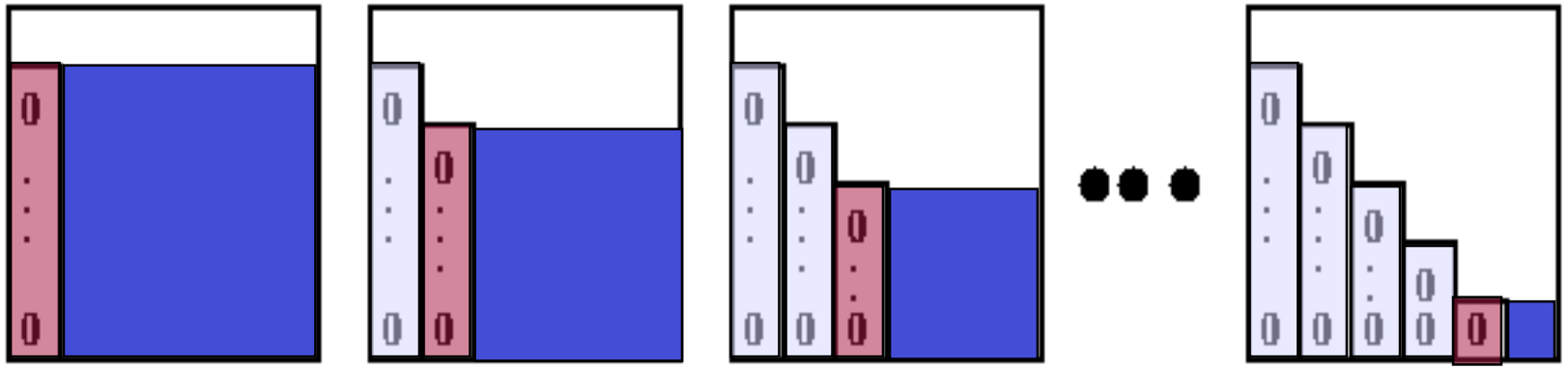
[0 -6 -5 -20] + -6*[0 -1 0 -4]
= [0 0 -5 4]

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & -6 & -5 & -20 \end{array}\right]$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & 0 & -5 & 4 \end{array}\right]$$

Overview

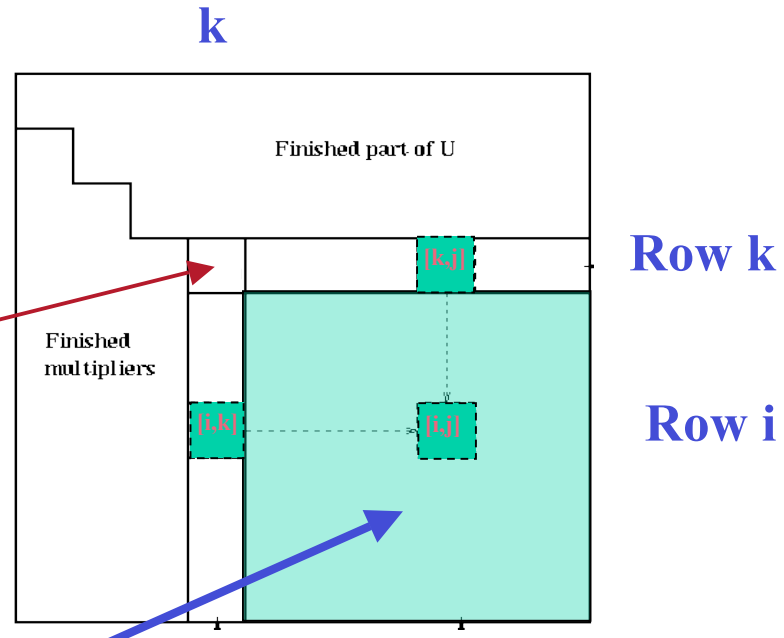
```
for k = 0 to n-1              // For each column k
   for i = k+1 to n-1         // Eliminate entries below the diagonal:
                              //    subtract a multiple of row k
                                   from  succeeding rows i

        A[i,k+1:n ] – = (A[i,k] / A[k,k]) * A[k, k+1:n ]

   end for
end for
```



Column 0 Column 1 Column 2 Column n-1

Eliminating the entries below the diagonal

- For each column **k : 0 to n-1**
- … subtract multiples of row **k**: **A[k,k+1:n]** …
  … from rows **i = k+1 to n**
- Multipliers **m_{ik} = A[i,k]/A[k,k]**
- … cancel the elements below the diagonal: **A[k+1:n-1,k]**
- We only update to the right of and below **A[k,k]**



for i = k+1 to n-1
A[i,k+1:n] – = m_{ik} × A[k, k+1:n]

Problems with roundoff

- The rank-1 update step uses division …

```
A[i, k+1:n ] -= ( A[i,k]/A[k,k]) * A[k,k+1:n]
```

- How to handle vanishing denominators or ones that are very small
- Gaussian elimination will fail with this matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- But we can avoid the problem if we swap rows

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Pivoting to avoid stability problems

- We call this process of swapping rows ***partial pivoting***
- Assume we carry 3 decimal digits of precision
- Consider the following A matrix and RHS b

$$A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- The correct solution is

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Notation

- Consider the following system of equations

$$
\begin{aligned}
x_0 + x_1 + x_2 &= 3 \\
4x_0 + 3x_1 + 4x_2 &= 8 \\
9x_0 + 3x_1 + 4x_2 &= 7
\end{aligned}
$$

- We usually write the system as an ***augmented matrix***

$$
\left[\begin{array}{ccc|c}
1 & 1 & 1 & 3 \\
4 & 3 & 4 & 8 \\
9 & 3 & 4 & 7
\end{array}\right]
$$

Roundoff

- Eliminate zero in row 2 by subtracting $10^4 \times$ row 0

$$L = \left[\begin{array}{ll|l} 10^{-4} & 1 & 1 \\ 0 & 1 - 10^4 & 2 - 10^4 \end{array}\right]$$

- But $1 - 10^4$ rounds to -10^4

$$L = \left[\begin{array}{ll|l} 10^{-4} & 1 & 1 \\ 0 & -10^4 & -10^4 \end{array}\right]$$

Stability problems due to roundoff

- Thus

$$L|b = \left[\begin{array}{ccc} 10^{-4} & 1 & 1 \\ 0 & -10^{4} & -10^{4} \end{array}\right]$$

- We now back substitute to solve for x_2 and then x_1

  $-10^4 x_2 = -10^4 \Rightarrow \mathbf{x_2 = 1}$

- Substituting the value of x_2 into the first equation

  $10^{-4} x_1 + 1 * x_2 = 1 \Rightarrow 10^{-4} x_1 = 0 \Rightarrow \mathbf{x_1 = 0}$

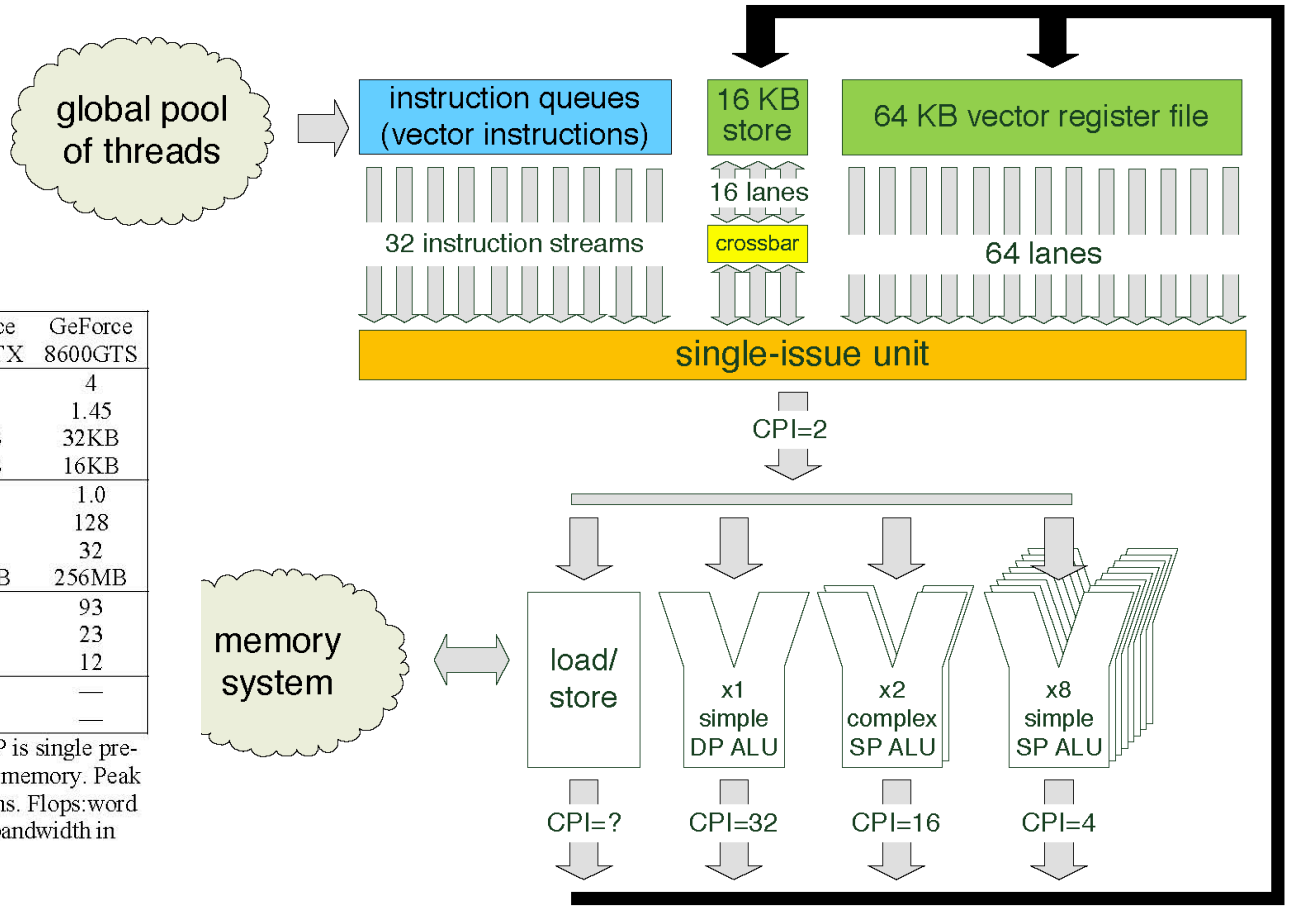
- **But the correct solution is $x_1 = x_2 = 1$**

Partial Pivoting

- Rule: pick the largest value in the column
- This is called partial pivoting, because only rows are swapped
- It can be shown that when with partial pivoting, we compute **A = P L U**, where P is a permutation matrix expressing the rows swaps
- We can also swap columns: ***full pivoting***
- But full pivoting is more expensive to implement

Props

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

SGEMM Code

```
__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

Volkov and Demmel

SGEMM Code Structure

```
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 B^T
   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.

Volkov and Demmel

Memory Latency

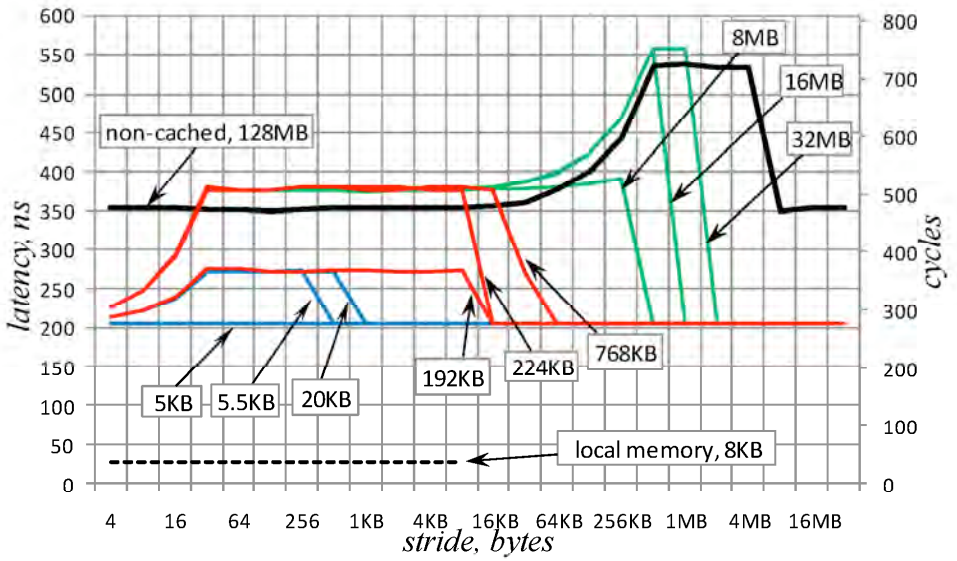


Figure 1: Memory latency as revealed by the pointer chasing benchmark on GeForce 8800 GTX for different kinds of memory accesses. Array size is shown in the boxes. Cached access assumed unless otherwise specified. Blue, red and green lines highlight 5KB cache, 192 KB cache, and 512KB memory pages respectively.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

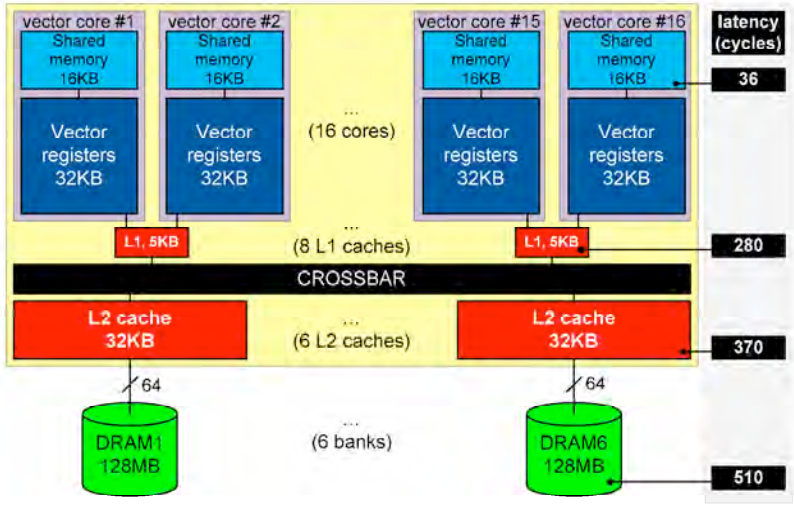


Figure 2: Summary of the memory system of 8800GTX according to our study. Sizes of the on-chip memory levels are shown in the same scale. Latencies shown are for the cached access. Note the small L1 caches and large register files.

Volkov and Demmel

Performance

Pipeline latency

Operation	Unit	GTX280
$a = a + b$, $a = a * b$	SP	24
same w/ b is in smem		26
$a = a * b + c$		24
same w/ b is in smem		28
$a = log2(\|a\|)$, $a = rsqrt(a)$	SFU	28
$a = a + b$, $a = a * b$	DP	48
$a = a * b + c$		52

$$Time = 4\mu s + \frac{bandwidth\ required}{127GB/s}$$

Memory Bandwidth

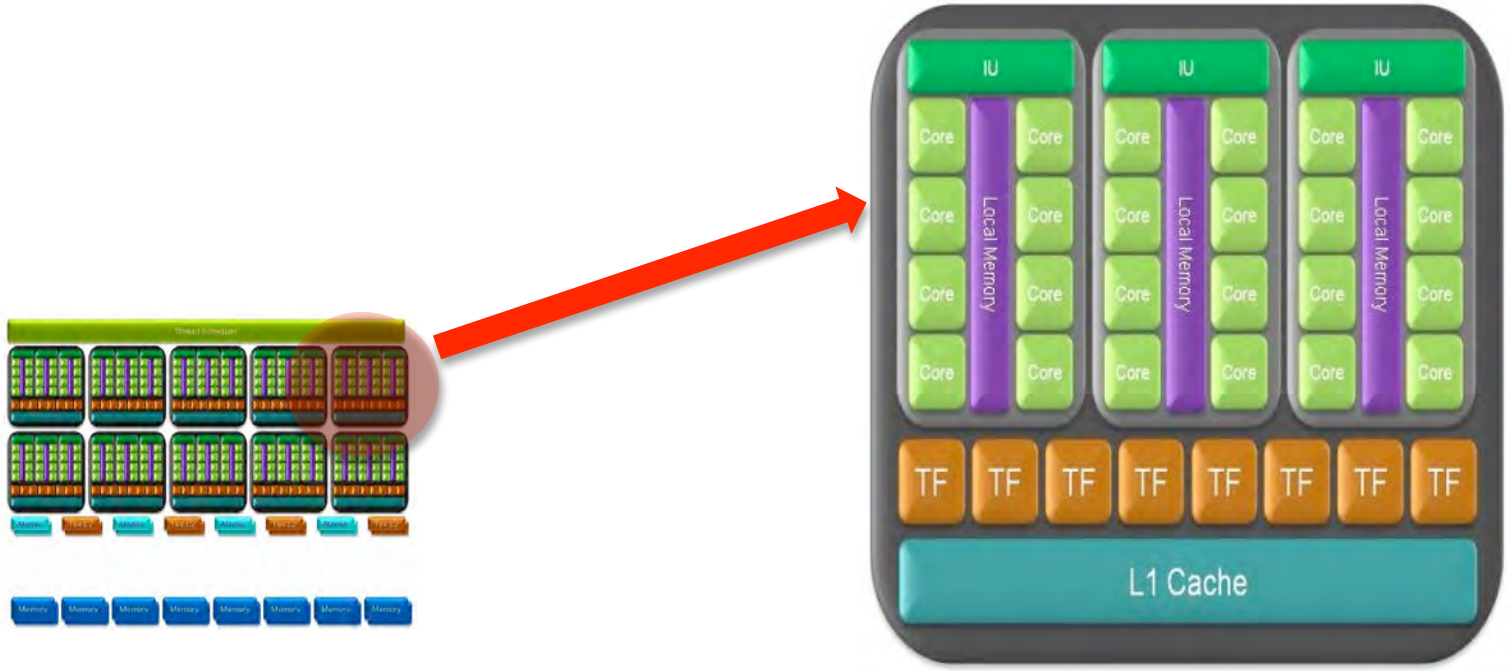
GPU	8800GTX	8600GTS	9800GTX	GTX280
at pins, GB/s	86	32	70	141
aligned copy	89%	83%	85%	89%
misaligned	9%	10%	9%	51%
stride-2	9%	10%	9%	45%
stride-10	10%	10%	9%	10%
stride-1000	0.9%	2.1%	1.1%	1.1%

Throughput using shared memory

Operation	8800GTX
$a+b*s[i]$	66%
$a+a*s[i]$	66%
$a+s[i]$	74%

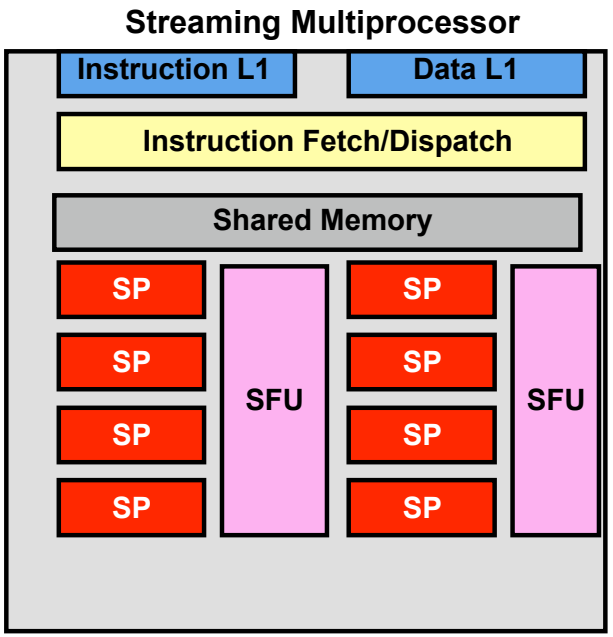
Volkov and Demmel

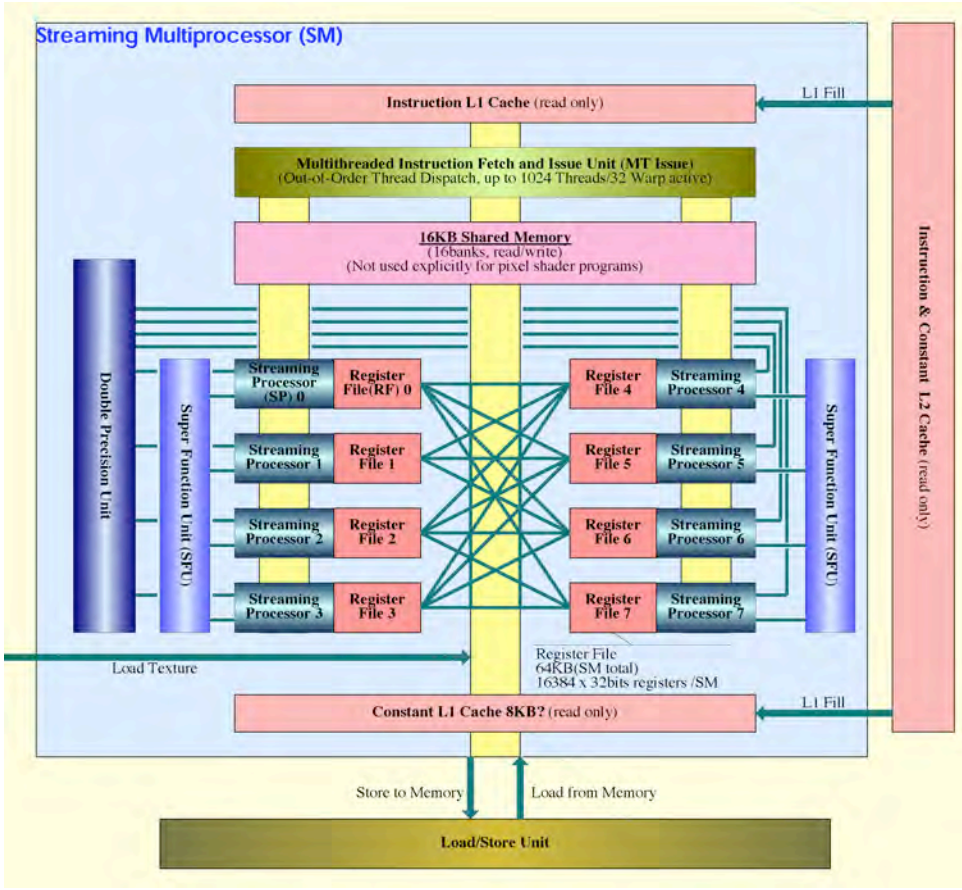
GeForce GTX 280



nVidia

Streaming Multiprocessor



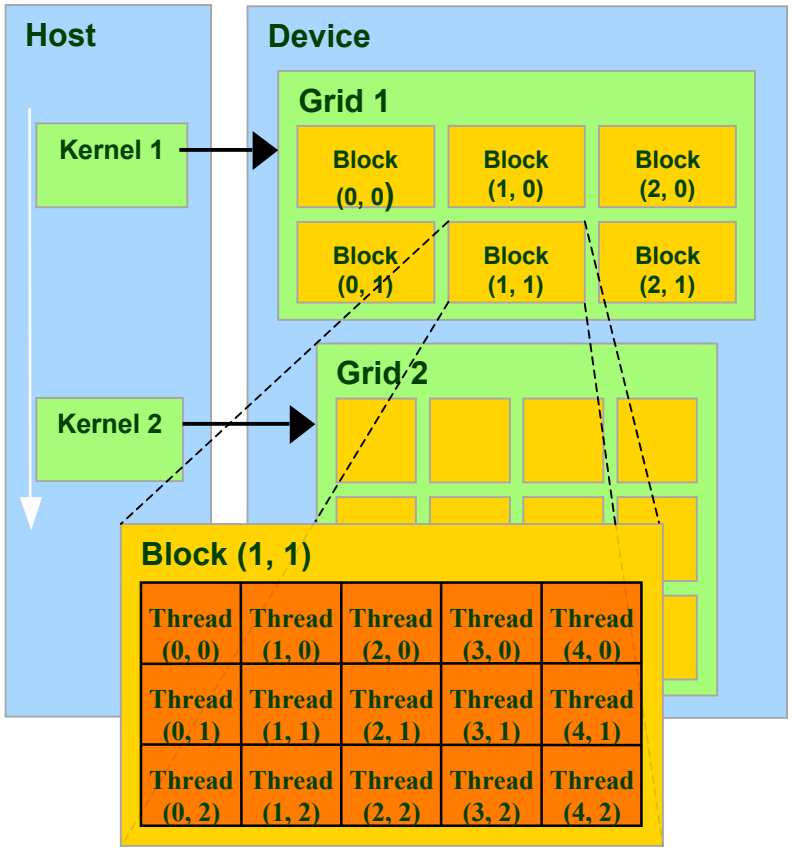


DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

H. Goto

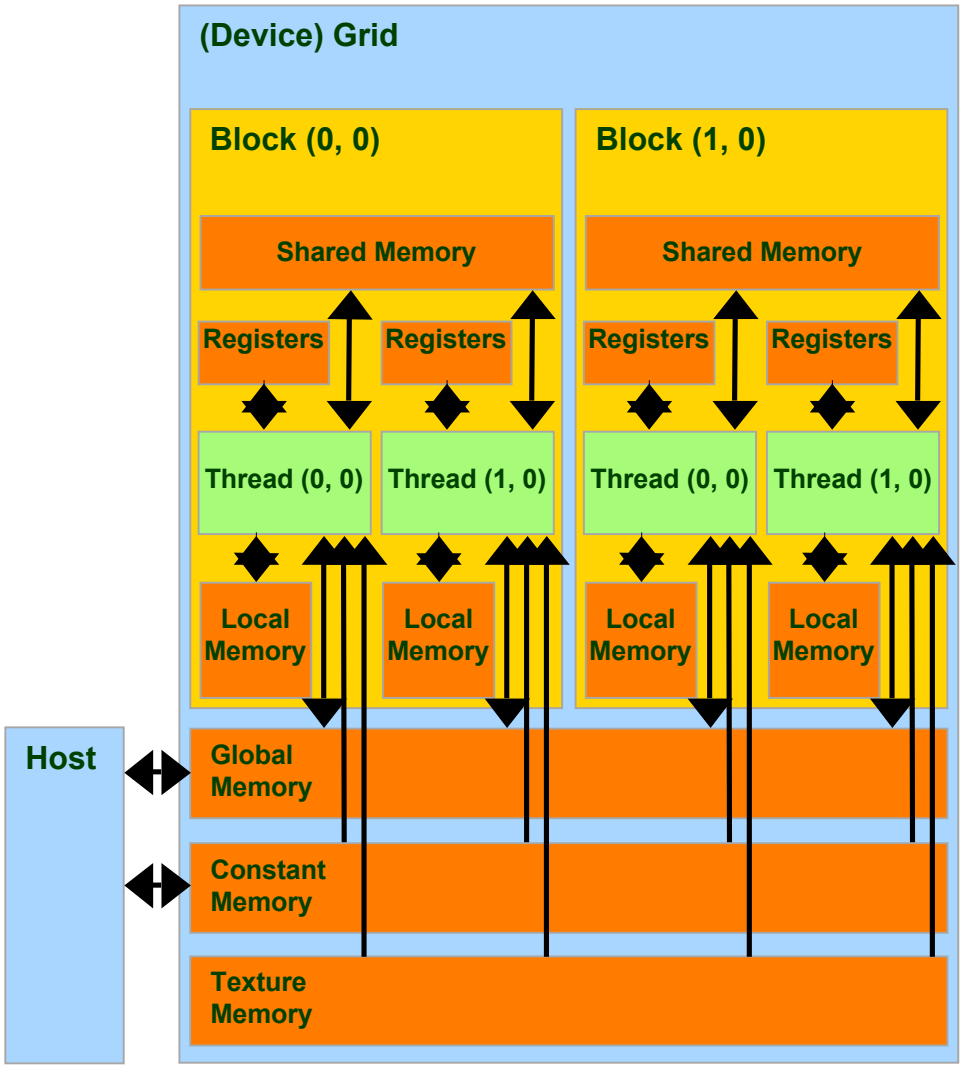
Hierarchical Thread Organization

Kernel<<<2,3>,<3,5>>>



DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

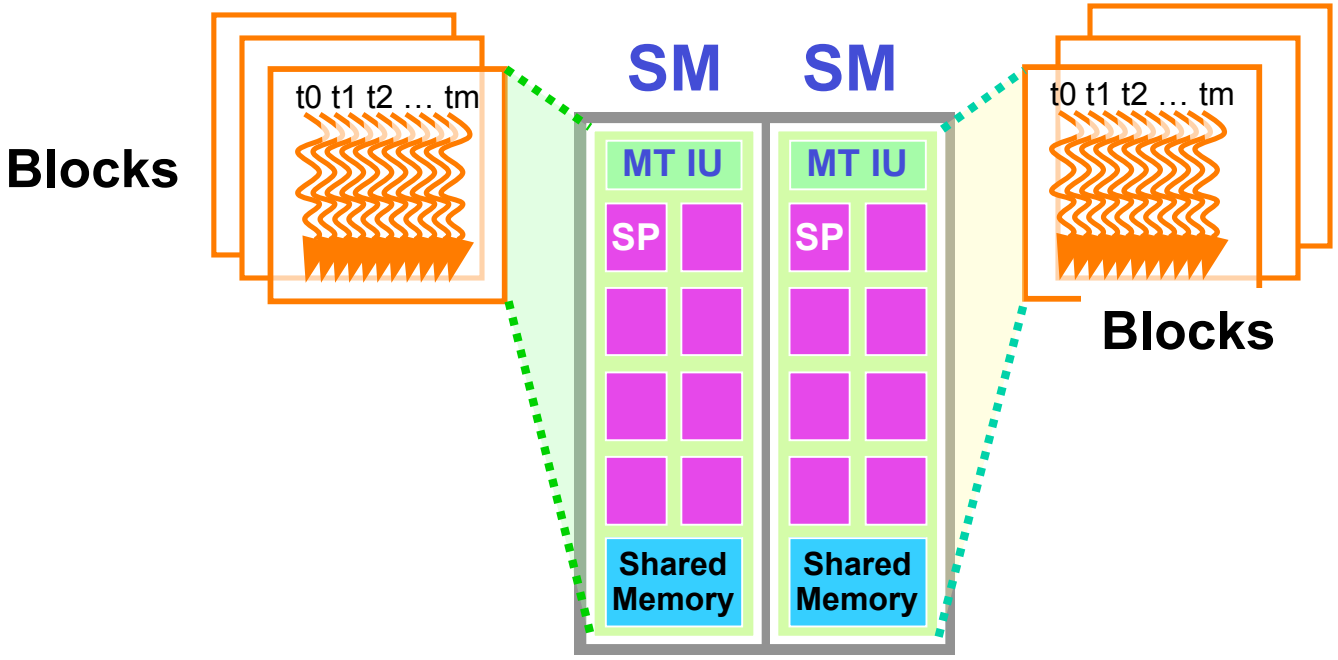
Memory Hierarchy



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	--

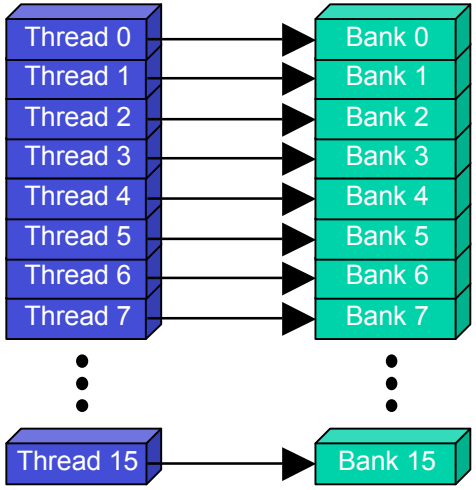
Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

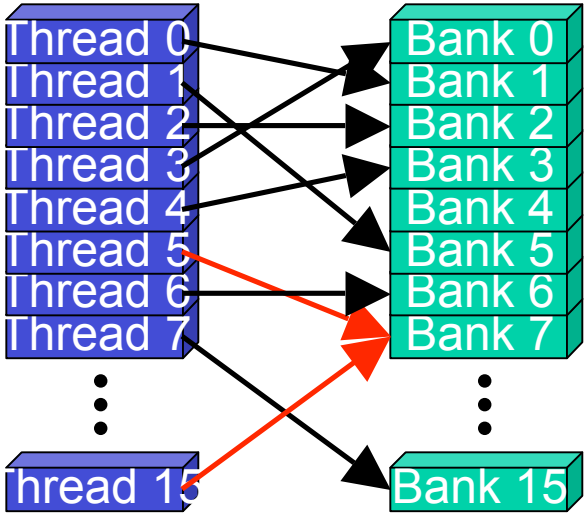
Occupancy



DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Bank Conflicts






```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = a[idx]+1.f;
```

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC