Lecture 9
CSE 260 – Parallel Computation (Fall 2015)

Scott B. Baden

Performance modeling

Further improvements to matrix multiplication

Today's lecture

- Performance modeling
- An improved matrix multiply

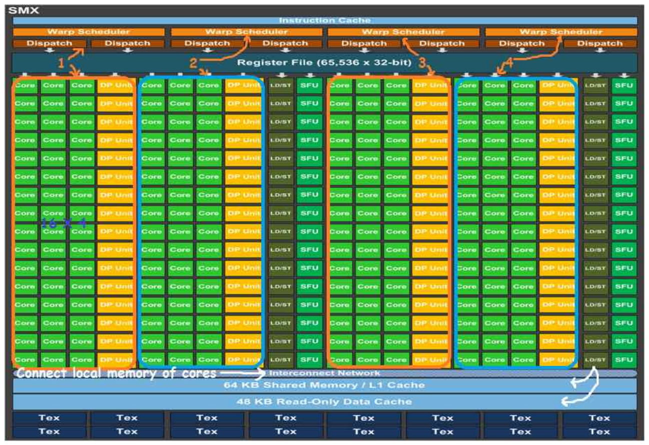
Performance modeling

- Given *N*, application flop rate, and peak rates of the hardware
  - Determine if app is compute bound or communication bound
  - Predict performance of unblocked algorithm and account for discrepancy with observation
- The naïve algorithm
  - N^3 multiply-adds
  - Without tiling, algorithm loads N^3 doubles precision words@ 8 bytes/word (we ignore C)
- The hardware
  - One GPU of the K80 can perform 832 MADs / cycle and transfer 240 GB/sec
  - Processor clock runs at 823.5 MHz

Tesla Kepler K80/K20m (GK 210/110)

- Sorken has device capability 3.7, Stampede has 3.5
  - 11¼ (5) GB device memory (frame buffer)@ 240 (208) GB/s
  - 1.5MB (1.25MB) shared L2 Cache (by all SMXs)
  - 13 SMXs (2496 cores) on Sorken and Stampede
- Sorken's K80 (GK210 GPU) has more registers and larger shared memory per device than Stampede's K20m (GK110 GPU)
  - 192 SP cores, 64 DP cores, 32 SFUs, 32 Load/Store units
  - Each scalar core: fused multiply adder, truncates intermediate result
  - 112K (64KB) on-chip memory configurable as scratchpad memory + L1 cache
  - 128K (64K) x 32-bit registers up to 255/thread
  - 1 FMA /cycle = 2 flops/cycle/ DP core*64 DP/SMX*13 SMX = 1664 flops/cyc @823.5 MHz (705.5 MHz) = 2.74 TFLOPS per GPU (1.17)

Nvidia

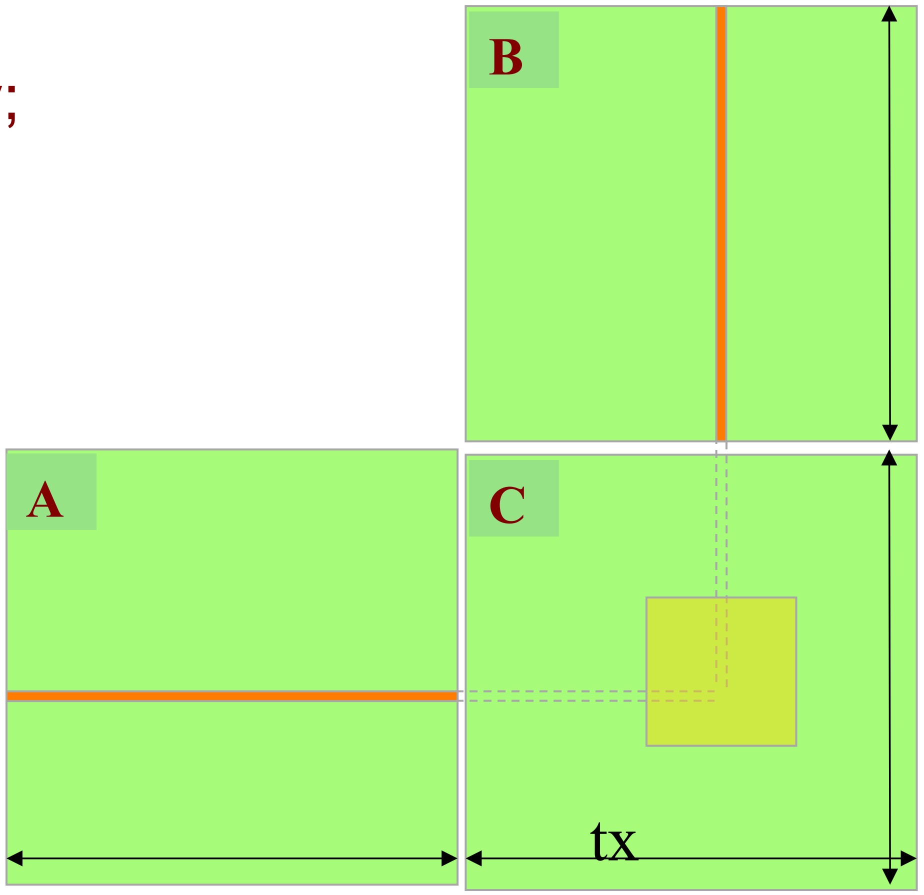

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
1
2
3
4
Core
DP Unit
LD/ST
SFU
Connect local memory of cores
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Analysis

- Based on work to be done, data to be moved, and hardware performance
  - Predicted data motion time: 89 **milli**seconds
  - Predicted computation time: 195 **micro**seconds
  - The application is **communication bound**
- The measured running time: 227ms (118GFlops)
- Why did we run about twice as slow?

Do memory accesses coalesce?

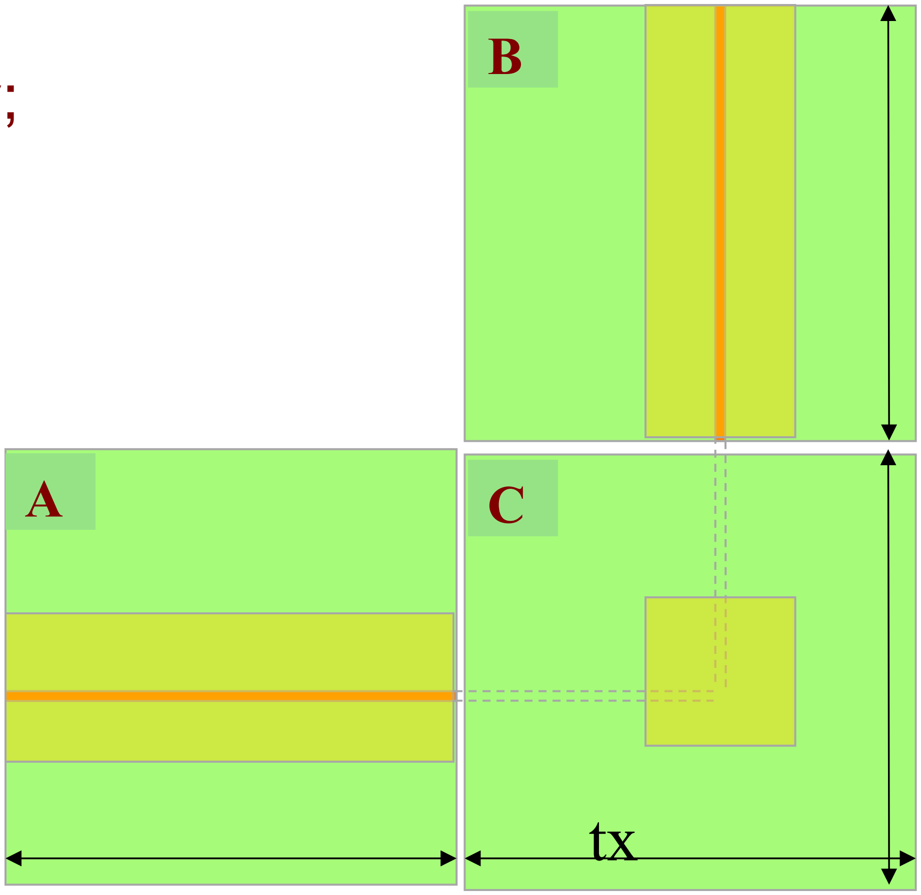
```
int I   =  by*blockDim.y + ty;
int J  =  bx*blockDim.x + tx;
int N =  blockDim.y*gridDim.y;
if ((I < N) && (J < N)){
    float _c = 0;
    for (k = 0; k < N; k++) {
        double a = A[I * N + k];
        double b = B[k * N + J];
        _c += a * b;
    }
    C[I * N + J] = _c;
 }
```



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Do memory accesses coalesce?

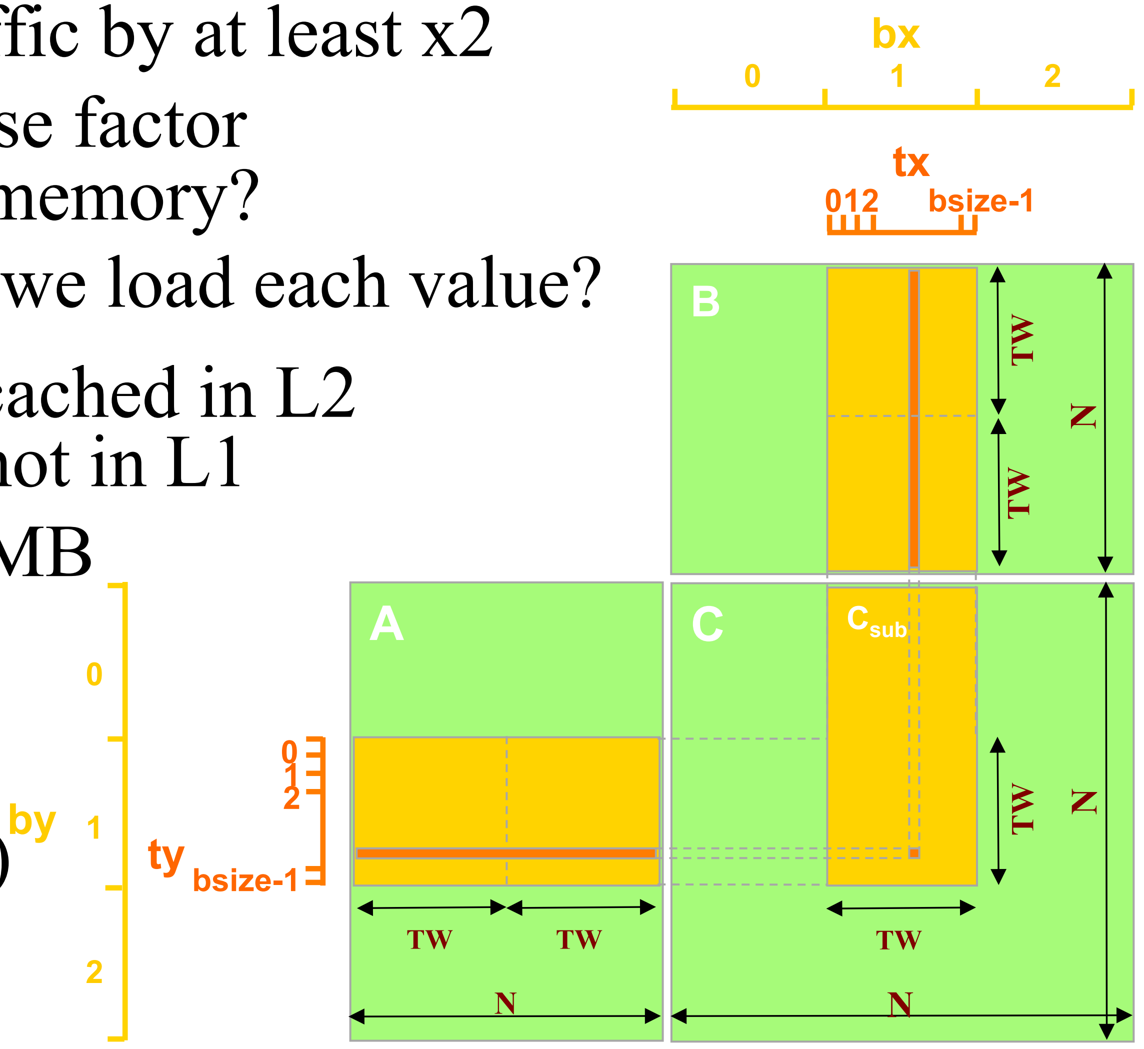
```
int I   =  by*blockDim.y + ty;
int J  =  bx*blockDim.x + tx;
int N =  blockDim.y*gridDim.y;
if ((I < N) && (J < N)){
     float _c = 0;
     for (k = 0; k < N; k++) {
         double a = A[I * N + k];
         double b = B[k * N + J];
         _c += a * b;
     }
     C[I * N + J] = _c;
   }
```



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Tiled algorithm

- Running time: 104 ms (259 GF): ~x2 faster
- Reduces memory traffic by at least x2
- Why not x32, the reuse factor realized with shared memory?
- How many times do we load each value?
- Coalesced accesses cached in L2 (1.5MB all SMXs), not in L1
- A block consumes 8MB in each of 13 SMXs (and 2 blocks/SMX)
- Each thread uses 30 registers (30K/block)
- There are many registers to spare!



Tiled Code

- Code on page 112 (some identifier name changes)

```
__global__ mmpy(double *A, double *B, double *C){
    __shared__ double A[TW][TW], A[TW][TW];
    int tx = threadIdx.y,   ty = threadIdx.x;
    int by = blockIdx.y,    bx = blockIdx.x
    int  I = by*TW + ty,     J = bx*TW+tx;
    double Cij  = 0;
    for (int kk=0; kk<N/TW; kk++){
        As[ty][tx] = A[I*N + kk*TW+tx];
        Bs[ty][tx] = B[(kk*TW+ty)*N + J];
        __syncthreads();
        for (int k=0; k<TW; k++)
           Cij+=  As[ty][k] * Bs[k][tx];
        __syncthreads();
     C[I*N + J] = Cij;
```

Today’s lecture

- Memory coalescing
- Avoiding bank conflicts
- Further Improvements to Matrix Multiply

How to improve matrix multiply still further

- Follows Volkov and Demmel, SC08
- Hide arithmetic latency using fewer threads
- Hide memory latency using fewer threads
- Improving performance using fewer threads
    - We can reduce number of threads through lower occupancy …
    - ..by making better use of registers we can trade locality against parallelism
- Code was implemented on a 1.x device so some details will be different (more registers on Kepler, for example)

Latency

- The time required to perform an operation
- The GK104 issues 1 instruction / cycle, the vector unit has 8 cores (SM): 4 cycles to issue a warp
- Instructions wait on dependencies
  x = a + b; // ~20 cycles to complete
  y = a + c; // independent, we start any time
- z = x + d; // dependent, wait on x

Arithmetic throughput

- The rate we perform an operation (flops/cycle)
- Arithmetic: 1.3TFlops/sec = 480 ops/cycle
- Memory: 177 GB/sec ~= 32x 32 bit loads per cycle

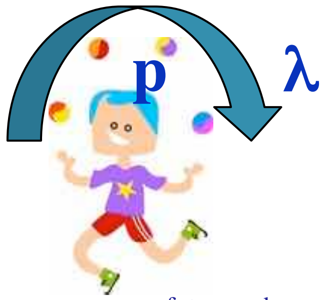
How do we hide latency?

- Do something else while waiting for an operation to complete
- This where Little's Law applies
- Required parallelism depends on latency and throughput

  # Parallelism (threads) = latency × throughput

  $T = \lambda \times p$

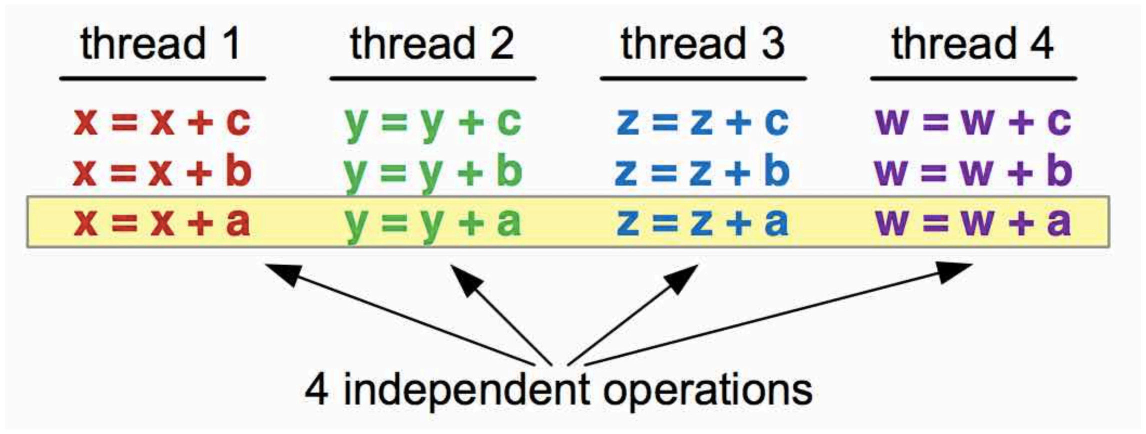
- Required parallelism depends on op; for single precision
  - GT200 (C1060): 24 CP * 8 cores / SM = 192 ops/SM
  - GF104 (GTX 460, Cseclass03-07): 18 CP * 48   = 864
  - GK110?
- If we can't realize the required parallelism we run at less peak performance

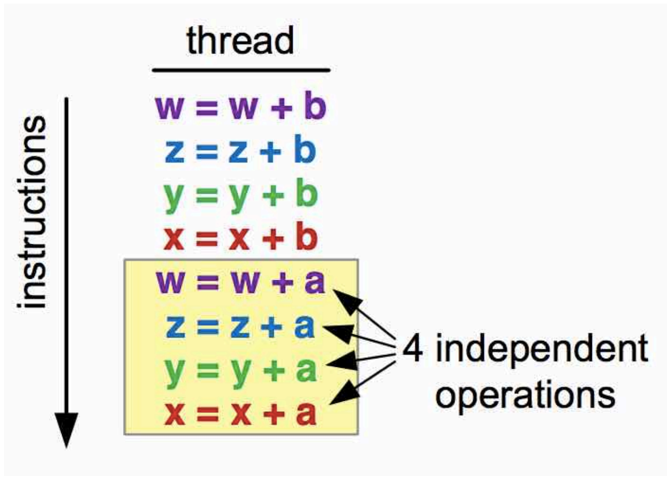


fotosearch.com

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 http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
- On GT200, 100% peak with 25% occupancy
  192 ops / cycle = 8 warps / 32 max possible warps





Courtesy V. Volkov, GTC-10

Hiding memory latency

- Parallelism = latency × throughput

  Arithmetic: 576 ops/SM = 18CP x 32/SM/CP
  Memory: 150KB = ~500CP (1100 nsec) × 150 GB/sec

- How can we keep 150KB in flight?
  - Multiple threads: ~35,000 threads @ 4B/thread
  - Do more work/thread (increase fetches per thread)
  - Larger fetches (64 or 128 bit/thread)
  - Higher occupancy

Copy 1 float /thread, need 100% occupancy

```
int indx = threadIdx.x + block * blockDim.x;
float a0 = src[indx];
dest[indx] = a0;
```

Copy 2 floats /thread, need 50% occ

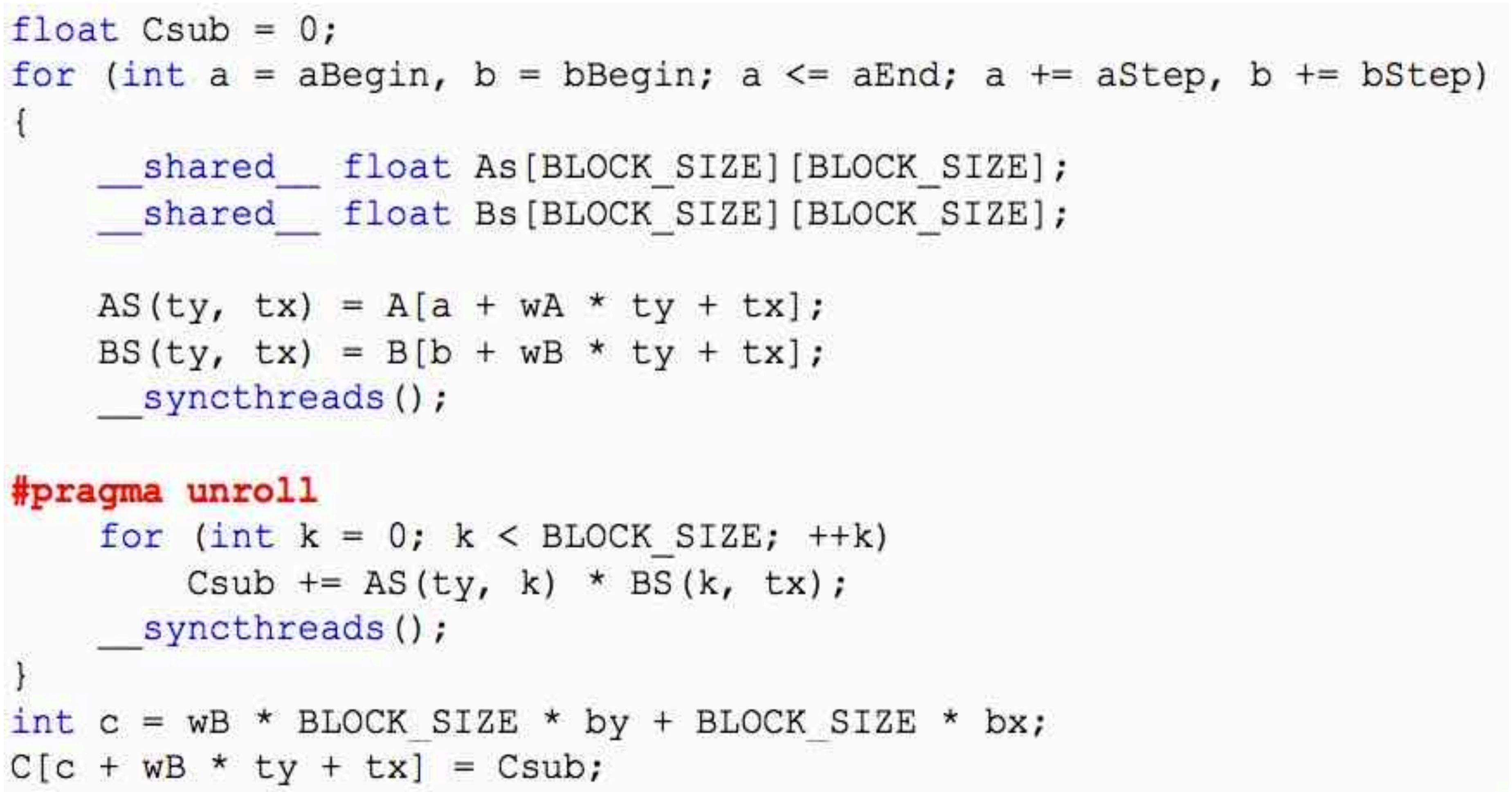
```
float a0 = src[indx];
float a1 = src[indx+blockDim.x];
dest[indx] = a0;
dst[index+blockDim.x] = a1;
```

Copy 4 floats /thread, need 25% occ

```
int indx = threadIdx.x + 4 * block * blockDim.x;
float a[4];  // in registers
for(i=0;i<4;i++) a[i]=src[indx+i*blockDim.x];
for(i=0;i<4;i++) dst[indx+i*blockDim.x]=a[i];
```

Incremental improvements to matrix multiply

- Follows V. Volkov [GTC10]
- From the book
- Gets 137 Gflops / sec



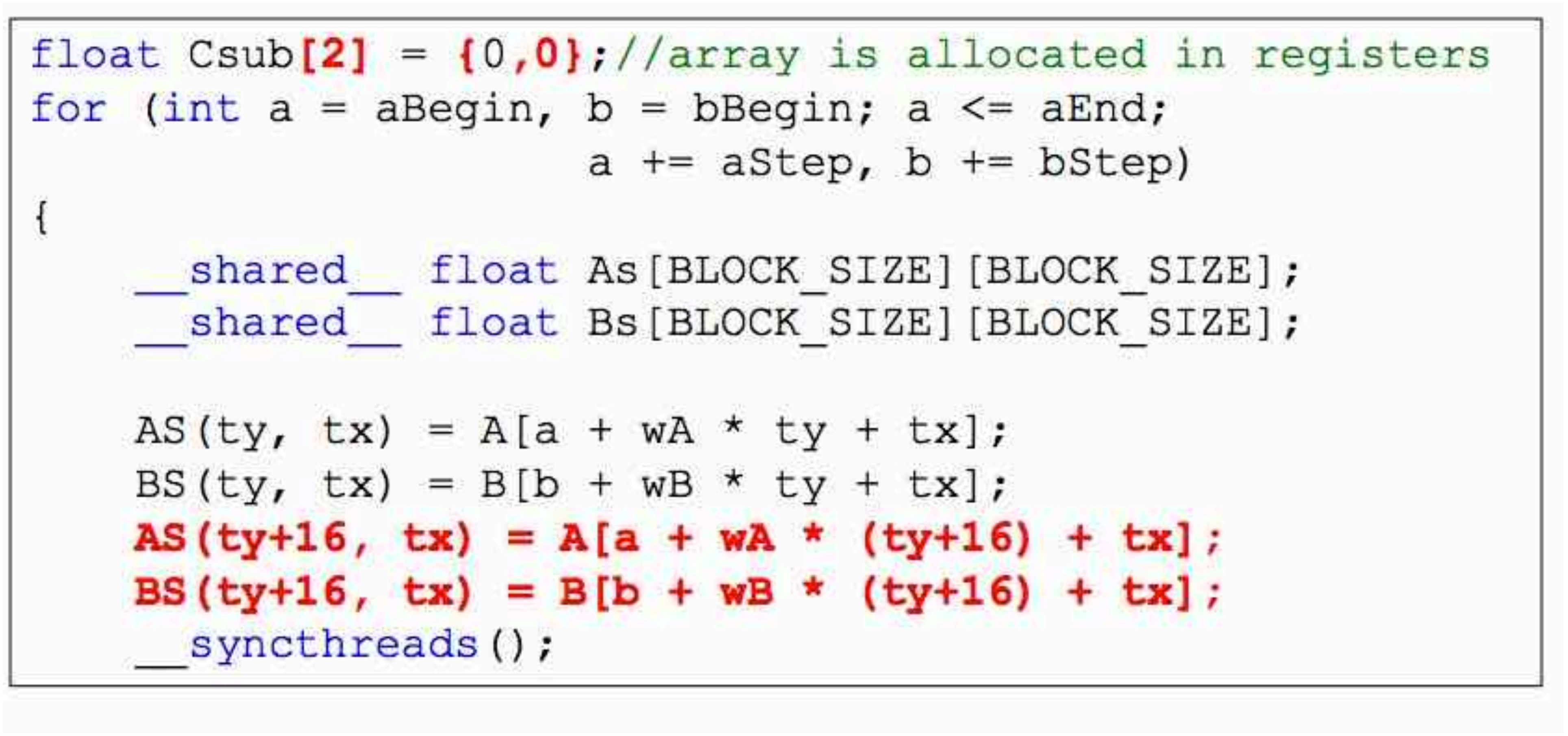
```
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Two outputs / thread

- 2 outputs, double the loads

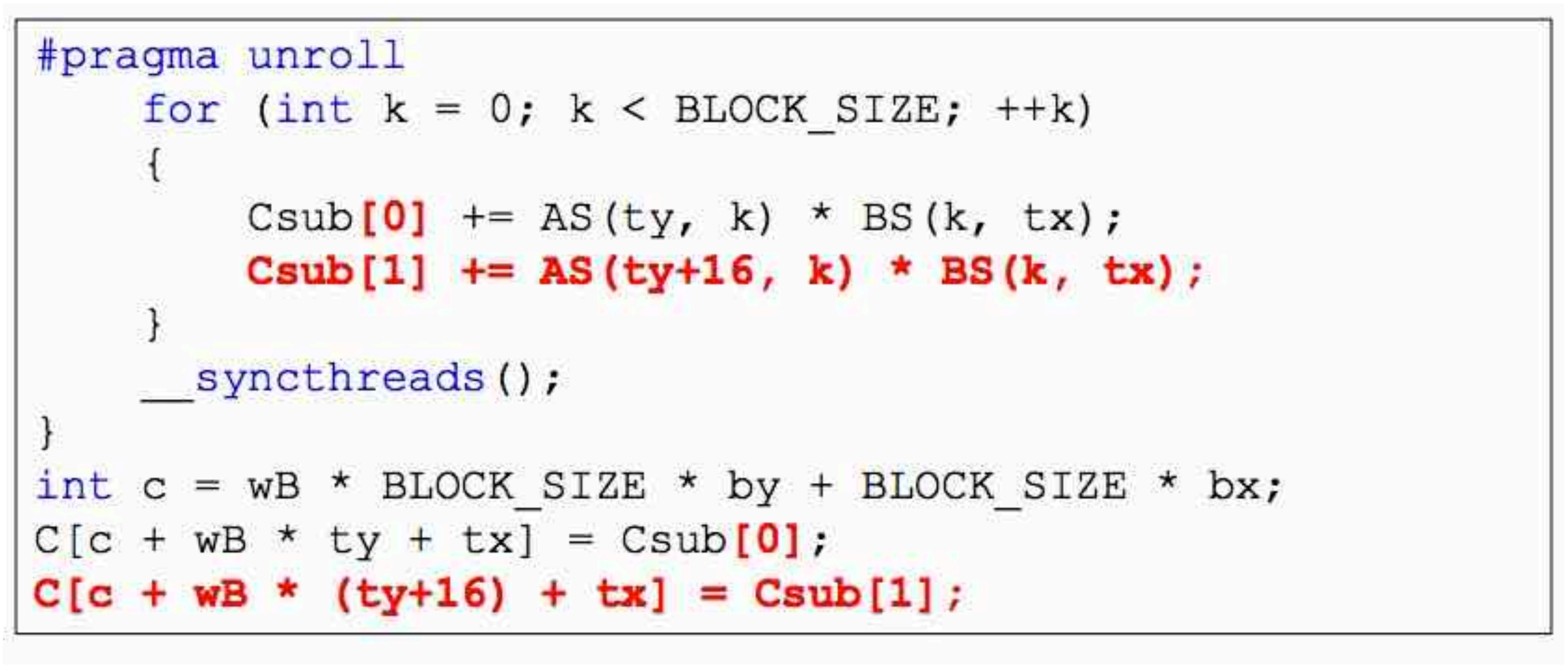


```
float Csub[2] = {0,0};//array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
                             a += aStep, b += bStep)
{
    __shared__  float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__  float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
```

Two outputs / thread, part 2

- ×2 flops and stores
- 341 Gflops/sec



```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+16, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```

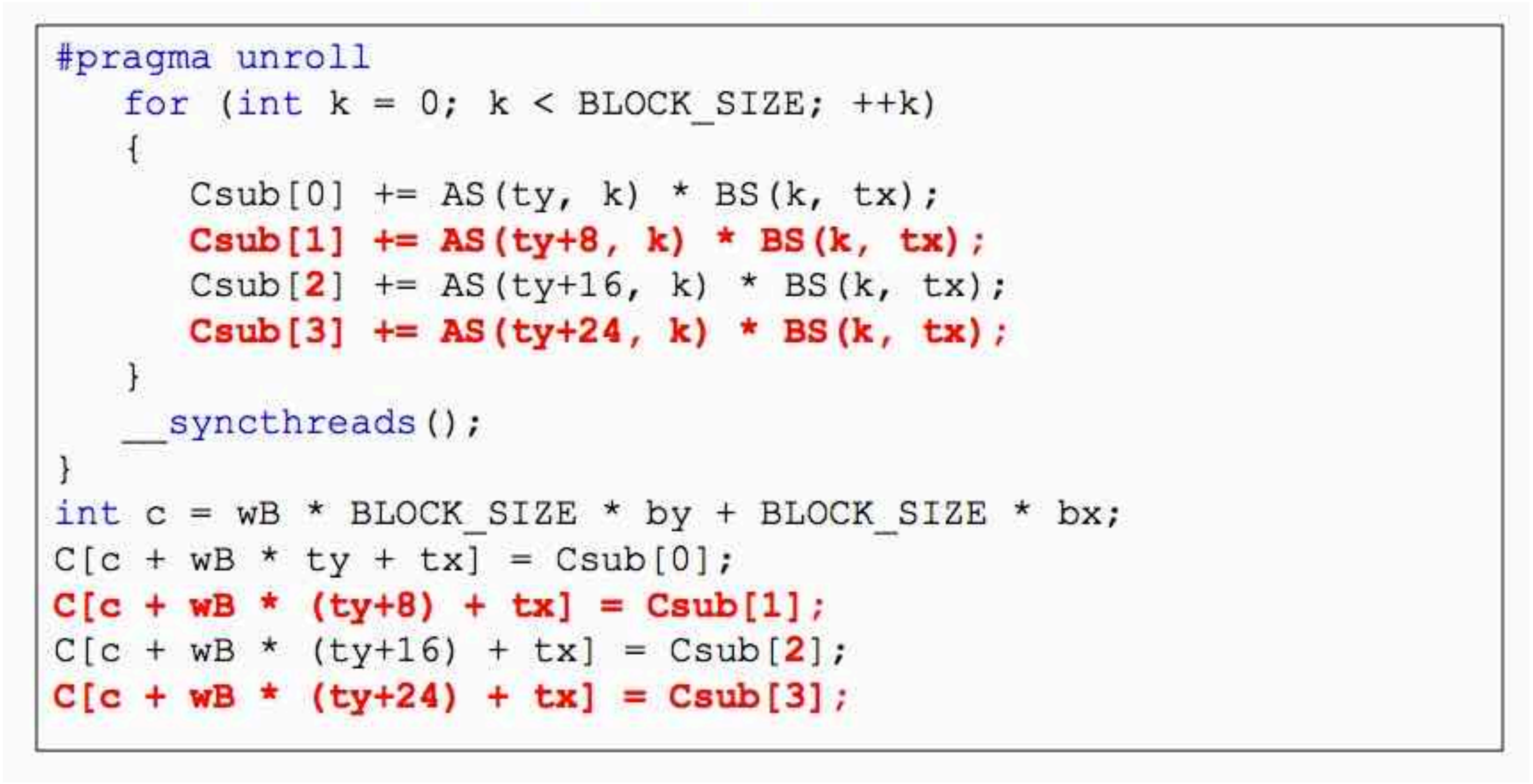
4 outputs / thread

```
float Csub[4] = {0,0,0,0};//array is in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
                        a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
```

4 outputs / thread

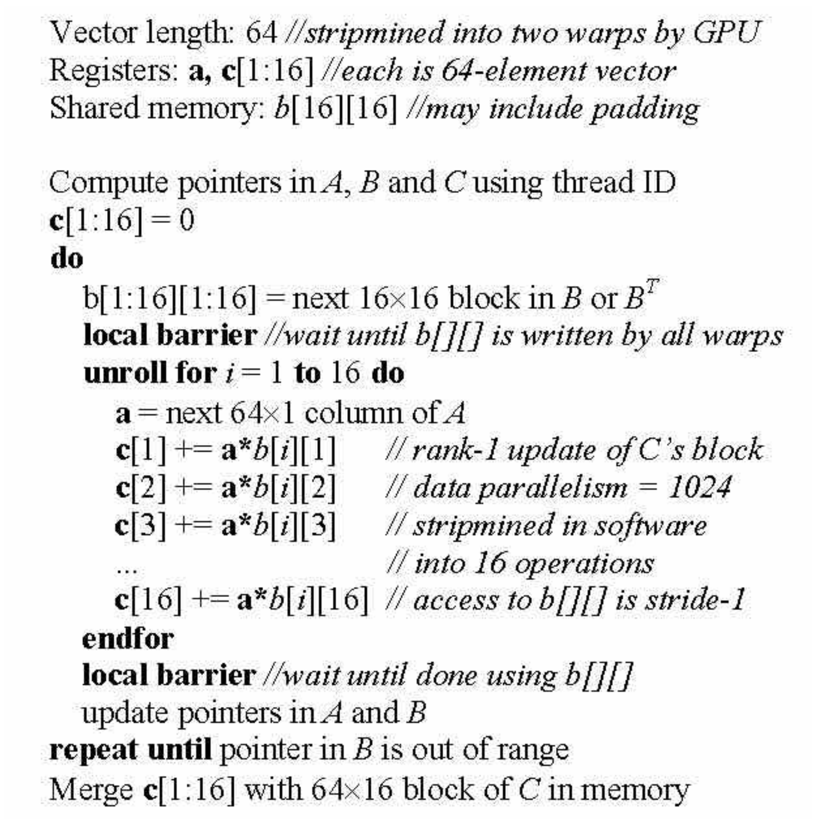
- 427 Gflops/sec [w/8 output/thread → 485 Gflops/s)
- ×2 # registers
- 50% occupancy



```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+8, k) * BS(k, tx);
        Csub[2] += AS(ty+16, k) * BS(k, tx);
        Csub[3] += AS(ty+24, k) * BS(k, tx);
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```

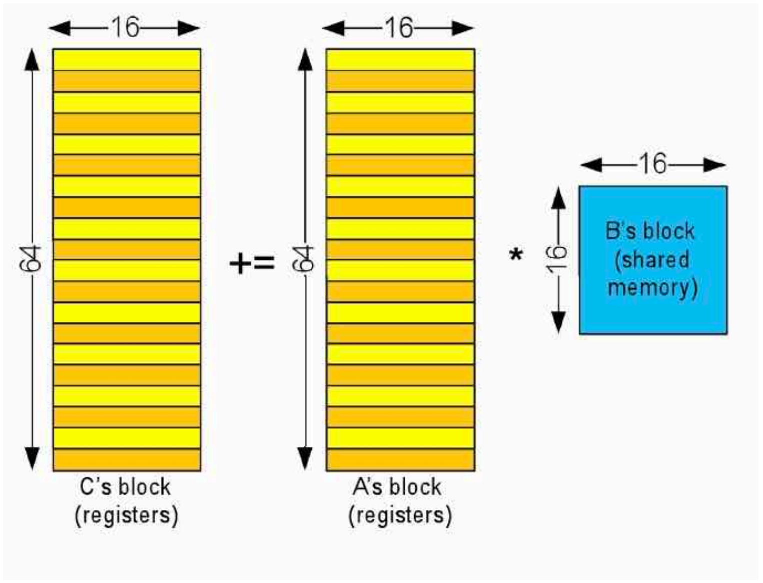
Volkov and Demmel's SGEMM

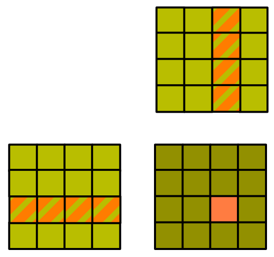
- Improve performance using fewer threads
  - Reducing concurrency frees up registers to trade locality against parallelism
  - ILP to increase processor utilization



```
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
```





Fin