Unrolling parallel loops

Vasily Volkov
UC Berkeley

November 14, 2011

Today

- Very simple optimization technique
- Closely resembles loop unrolling
- Widely used in high performance codes

Mapping to GPU: it starts with a loop

```
for( int i = 0; i < n; i++ )
    a[i] = b[i] + c[i];
```

```
__global__ void add( int *a, int *b, int *c )
{
   int i = threadIdx.x + blockIdx.x * gridDim.x;
   a[i] = b[i] + c[i];
}
```

GPU kernel

One loop iteration is mapped to one GPU thread

What if you unroll the loop before mapping?

Unroll the loop first...

```
for( int i = 0; i < n; i++ )
    a[i] = b[i] + c[i];
```

```
for( int i = 0; i < n; i += 2 )
{
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
}
```

2x fewer iterations, 2x more work per iteration

…and then map to GPU?

```
__global__ void add( int *a, int *b, int *c )
{
    int i = 2*(threadIdx.x+blockIdx.x*gridDim.x);
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
}
```

GPU kernel

2x fewer threads, 2x more work per thread

But why would you ever do that?

Agenda:

I. Speedup in molecular dynamics kernel

II. Speedup in radio astronomy kernel

III. Case study: a linear algebra kernel

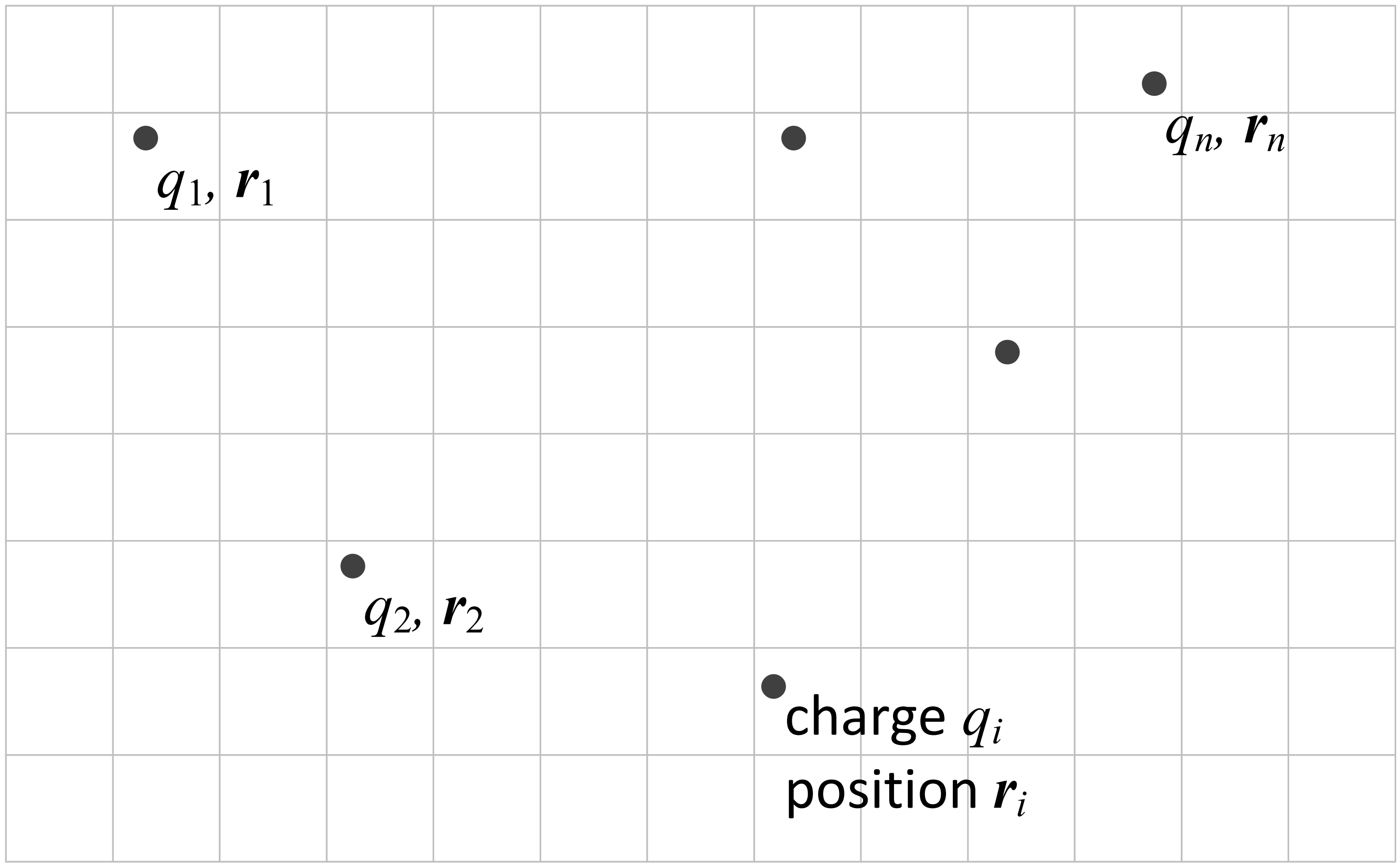
Example: molecular dynamics

One of the first works in CUDA:

– Stone et al. 2007. **Accelerating molecular modeling applications with graphics processors**, *JCC 28*, 16, 2618–2640.

Found that 8x “unrolling” gives 2x speedup

Charged particles on 3D grid



Goal:

Compute electric potential $V(\boldsymbol{r}) = \dfrac{1}{4\pi\epsilon_0}\sum \dfrac{q_i}{|\boldsymbol{r}_i - \boldsymbol{r}|}$ on grid

Pseudo-code for the problem

```
for each grid point i:
    for each particle j:
        add up j's contribution to i
    store the result
```

Parallelize the outer loop

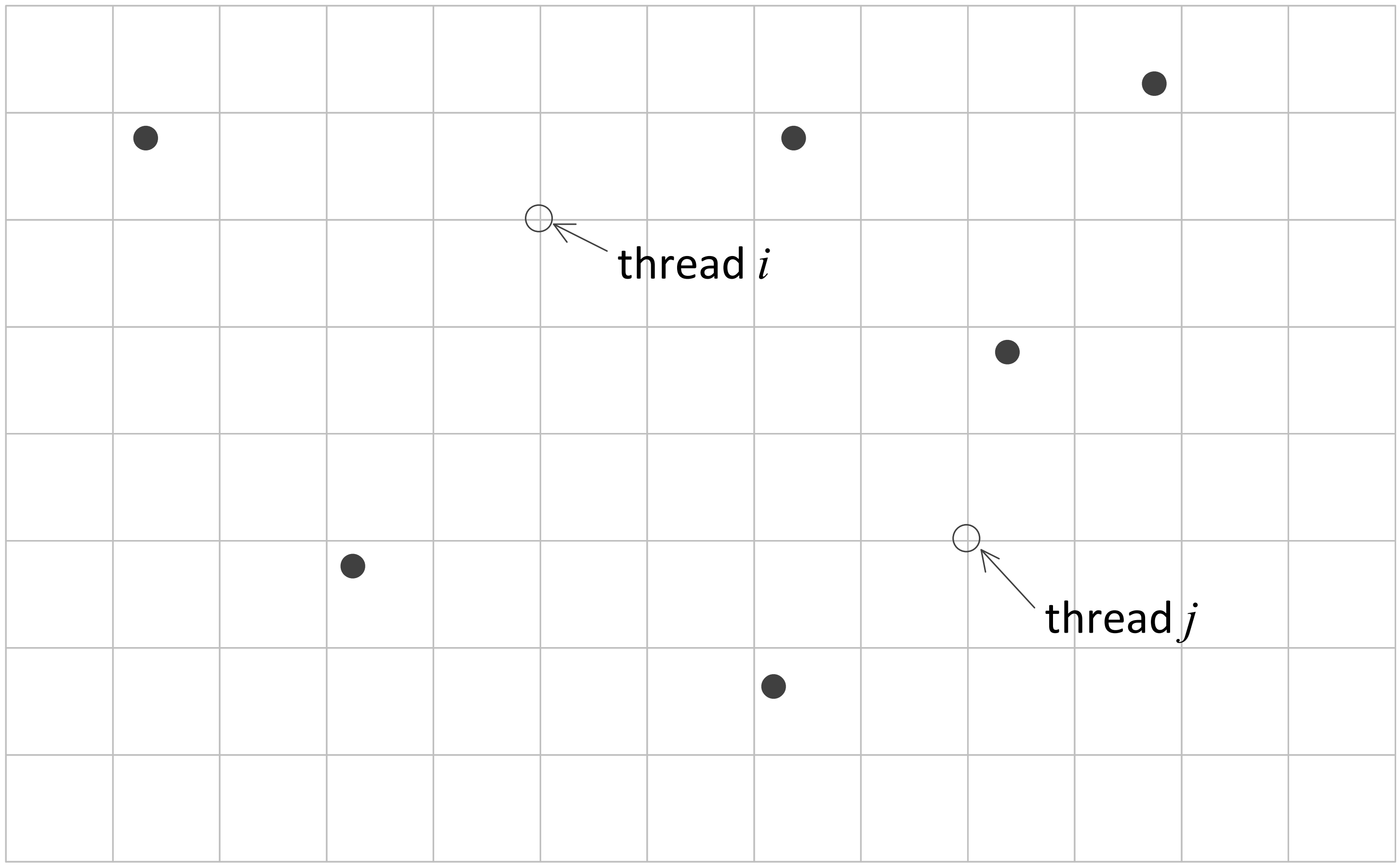
for each grid point `i` in parallel:

for each particle `j`:

add up `j`'s contribution to `i`

store the result

Mapping: one grid point per thread



19x faster than optimized CPU code

–Can we do better?

Unroll the parallel loop

for every second grid point `i` in parallel:

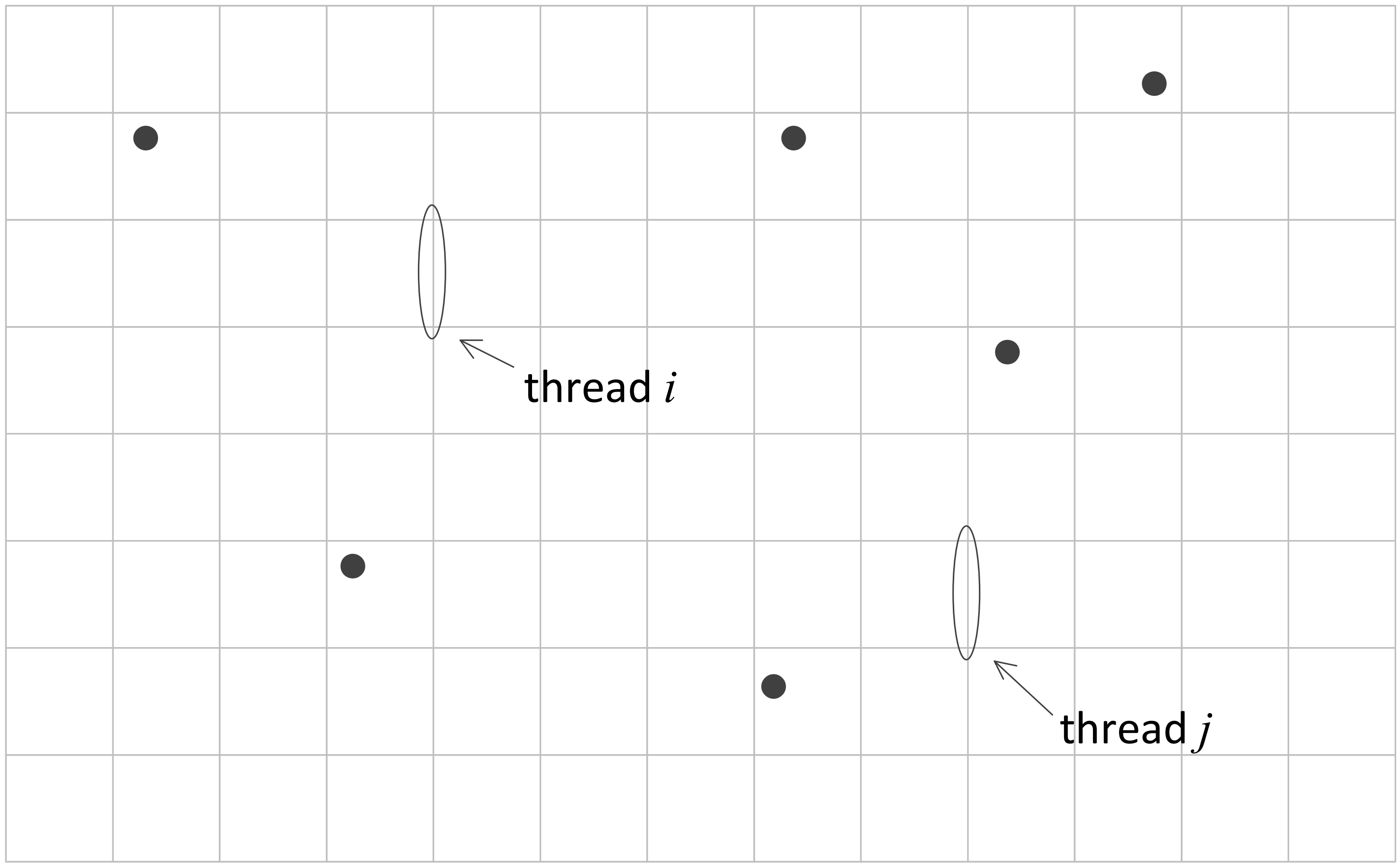
    for each particle `j`:

        add up `j`’s contribution to `i`

        add up `j`’s contribution to `i+1`

    store the both results

Multiple grid points per thread



Advantage: read q_i, $\boldsymbol{r}_i$ once, use multiple times

Also, can eliminate common subexpressions in $|\boldsymbol{r}_i - \boldsymbol{r}|$

“Unrolling” results in 2.2x speedup

Grid points per thread	Speedup vs quad core CPU
1	19
4	39
8	41

A substantial speedup for a simple optimization!

Radio astronomy

One of the later papers:

– Clark et al. 2011. **Accelerating Radio Astronomy Cross-Correlation with Graphics Processing Units**, *arXiv:1107.4264v2.*

1.8x speedup by doing 4x more work per thread

Array of antennas

Many antennas work as a single telescope

- For the cost of extra processing power

Input data: signal $X_i(t)$ from each antenna

Output: cross-correlation $S_{ij} = \sum_t X_i(t) X_j^\dagger(t)$

Different frequencies are processed separately

Parallelized pseudo-code

for each pair of antennas `i` and `j` in parallel:

for each time sample t:

$$S_{ij} := S_{ij} + X_i(t) X_j^*(t)$$

store the result

Unrolling the loops

for each pair of even `i` and `j` in parallel:

for each time sample t:

$$S_{ij} := S_{ij} + X_i(t) X_j^*(t)$$
$$S_{i+1,j} := S_{i+1,j} + X_{i+1}(t) X_j^*(t)$$
$$S_{i,j+1} := S_{i,j+1} + X_i(t) X_{j+1}^*(t)$$
$$S_{i+1,j+1} := S_{i+1,j+1} + X_{i+1}(t) X_{j+1}^*(t)$$

store the result

(mapped to threads)

“Unrolling” results in 1.8x speedup

Matrix entries per thread	Gflop/s
1x1	562
2x1	852
2x2	1023

Reason: data reuse in local variables

Blocking in GPU matrix multiply was used before CUDA, see: Moravánszky, A. 2003. **Dense Matrix Algebra on the GPU**.

Case study: Small linear solves

- Solve many independent 32x32 s.p.d. systems $Ax=b$
  - Solve one system per thread block
- Minimum flop solution: Cholesky+triangular solve
  - Challenging to implement efficiently in SIMD
- Use Gauss-Jordan instead, no pivoting
  - Drawback: does 6x more flops than Cholesky
- Here: omit right-hand side
  - Easy to add back with little overhead (1.2x slowdown)
- Target platform: GTX480, CUDA 4.0

Baseline solution

```
__shared__ float A[32][32];
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    //copy matrix to shared memory
    A[y][x] = in[32*32*problem+32*y+x];

    //run Gauss-Jordan in shared memory (see next slide)
    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i ) A[y][x] /= A[i][i];
        __syncthreads( );
        if( y != i ) A[y][x] -= A[y][i]*A[i][x];
    }

    //copy result to global memory
    out[32*32*problem+32*y+x] = A[y][x];
}
```

Gauss-Jordan in shared memory

1. Scale the pivot row

1	0	0	x	x	x	x	x
0	1	0	x	x	x	x	x
0	0	1	x	x	x	x	x
0	0	0	**X**	x	x	x	x
0	0	0	x	x	x	x	x
0	0	0	x	x	x	x	x
0	0	0	x	x	x	x	x
0	0	0	x	x	x	x	x

1	0	0	x	x	x	x	x
0	1	0	x	x	x	x	x
0	0	1	x	x	x	x	x
0	0	0	**1**	x	x	x	x
0	0	0	x	x	x	x	x
0	0	0	x	x	x	x	x
0	0	0	x	x	x	x	x
0	0	0	x	x	x	x	x

Get **1** on diagonal

2. Subtract it from every other row

1	0	0	**X**	x	x	x	x
0	1	0	**X**	x	x	x	x
0	0	1	**X**	x	x	x	x
0	0	0	1	x	x	x	x
0	0	0	**X**	x	x	x	x
0	0	0	**X**	x	x	x	x
0	0	0	**X**	x	x	x	x
0	0	0	**X**	x	x	x	x

1	0	0	**0**	x	x	x	x
0	1	0	**0**	x	x	x	x
0	0	1	**0**	x	x	x	x
0	0	0	1	x	x	x	x
0	0	0	**0**	x	x	x	x
0	0	0	**0**	x	x	x	x
0	0	0	**0**	x	x	x	x
0	0	0	**0**	x	x	x	x

Get **0** off diagonal

```
for( int i = 0; i < 32; i++ ) {
    if( y == i ) A[y][x] /= A[i][i];
    __syncthreads( );
    if( y != i ) A[y][x] -= A[y][i]*A[i][x];
    //no __syncthreads( ) needed here
}
```

Unroll the parallel loop

- Use half as many threads
- But twice as much work per thread
- This amounts to replicating lines of code

Unrolling 2x (red is new)

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    //copy matrix to shared memory
    A[2*y+0][x] = in[32*32*problem+32*(2*y+0)+x];
    A[2*y+1][x] = in[32*32*problem+32*(2*y+1)+x];

    //Gauss-Jordan in shared memory
    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/2 ) A[i][x] /= A[i][i];
        __syncthreads( );
        if( 2*y+0 != i ) A[2*y+0][x] -= A[i][x]*A[2*y+0][i];
        if( 2*y+1 != i ) A[2*y+1][x] -= A[i][x]*A[2*y+1][i];
    }

    //store the result in global memory
    out[32*32*problem+32*(2*y+0)+x] = A[2*y+0][x];
    out[32*32*problem+32*(2*y+1)+x] = A[2*y+1][x];
}
```

Unrolling 4x

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    A[4*y+0][x] = in[32*32*problem+32*(4*y+0)+x];
    A[4*y+1][x] = in[32*32*problem+32*(4*y+1)+x];
    A[4*y+2][x] = in[32*32*problem+32*(4*y+2)+x];
    A[4*y+3][x] = in[32*32*problem+32*(4*y+3)+x]; // do 4x more work

    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/4 ) A[i][x] /= A[i][i];
        __syncthreads( );
        if( 4*y+0 != i ) A[4*y+0][x] -= A[i][x]*A[4*y+0][i];
        if( 4*y+1 != i ) A[4*y+1][x] -= A[i][x]*A[4*y+1][i];
        if( 4*y+2 != i ) A[4*y+2][x] -= A[i][x]*A[4*y+2][i];
        if( 4*y+3 != i ) A[4*y+3][x] -= A[i][x]*A[4*y+3][i];
    }
    out[32*32*problem+32*(4*y+0)+x] = A[4*y+0][x];
...
```

Same but shorter

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    for( int j = 4*y; j < 4*(y+1); j++ )    // unrolled by compiler
        A[j][x] = in[32*32*problem+32*j+x];

    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/4 ) A[i][x] /= A[i][i];
        __syncthreads( );
        for( int j = 4*y; j < 4*(y+1); j++ )
            if( j != i ) A[j][x] -= A[j][i]*A[i][x];
    }

    for( int j = 4*y; j < 4*(y+1); j++ )
        out[32*32*problem+32*j+x] = A[j][x];
}
```

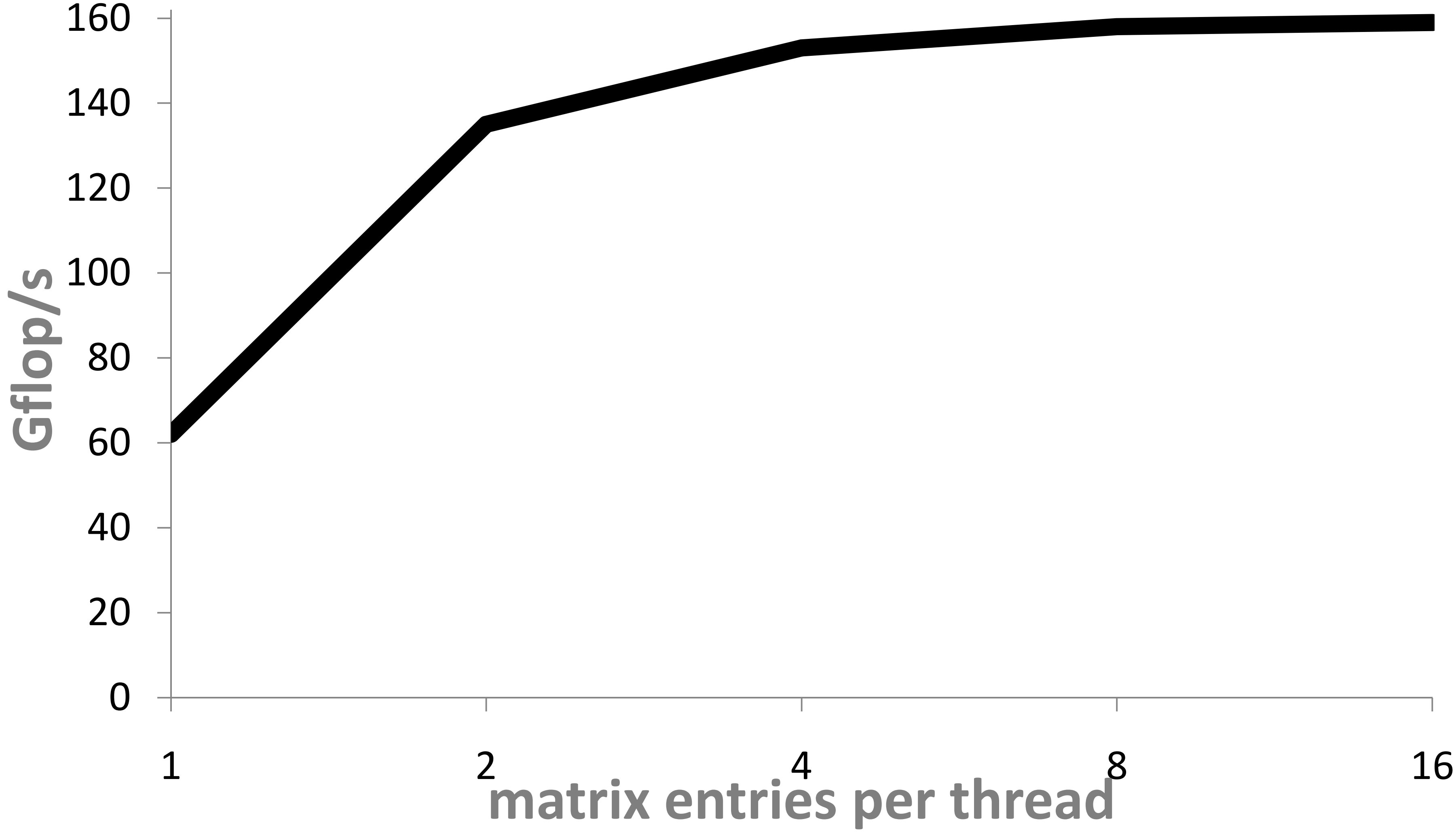
Unrolling 8x

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    for( int j = 8*y; j < 8*(y+1); j++ )
        A[j][x] = in[32*32*problem+32*j+x];

    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/8 ) A[i][x] /= A[i][i];
        __syncthreads( );
        for( int j = 8*y; j < 8*(y+1); j++ )
            if( j != i ) A[j][x] -= A[j][i]*A[i][x];
    }

    for( int j = 8*y; j < 8*(y+1); j++ )
        out[32*32*problem+32*j+x] = A[j][x];
}
```

Unrolling 16x

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    for( int j = 16*y; j < 16*(y+1); j++ )
        A[j][x] = in[32*32*problem+32*j+x];

    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/16 ) A[i][x] /= A[i][i];
        __syncthreads( );

        #pragma unroll // have to be explicit for heavy unrolling
        for( int j = 16*y; j < 16*(y+1); j++ )
            if( j != i ) A[j][x] -= A[j][i]*A[i][x];
    }

    for( int j = 16*y; j < 16*(y+1); j++ )
        out[32*32*problem+32*j+x] = A[j][x];
}
```

Aggregate speedup: 2.6x



Let’s use profiler to figure out what happened

Profiler statistics per thread

Elements per thread	Gflop/s	Registers per thread	Instructions executed per thread	Occupancy
1	62	8	397	0.67
2	135	9	470	1.00
4	153	11	783	1.00
8	158	15	1431	0.67
16	159	21	2740	0.33

– More resources consumed per thread

– Occupancy goes up and down

Doesn’t really explain the speedup

Profiler statistics per thread block

Threads per block	Gflop/s	Registers per block	Instructions per thread block	Thread blocks per multiprocessor
1024	62	8192	12704	1
512	135	4608	7520	3
256	153	2816	6264	6
128	158	1920	5724	8
64	159	1344	5480	8

Fewer resources used per thread block

- i.e. per same amount of work

More concurrent thread blocks

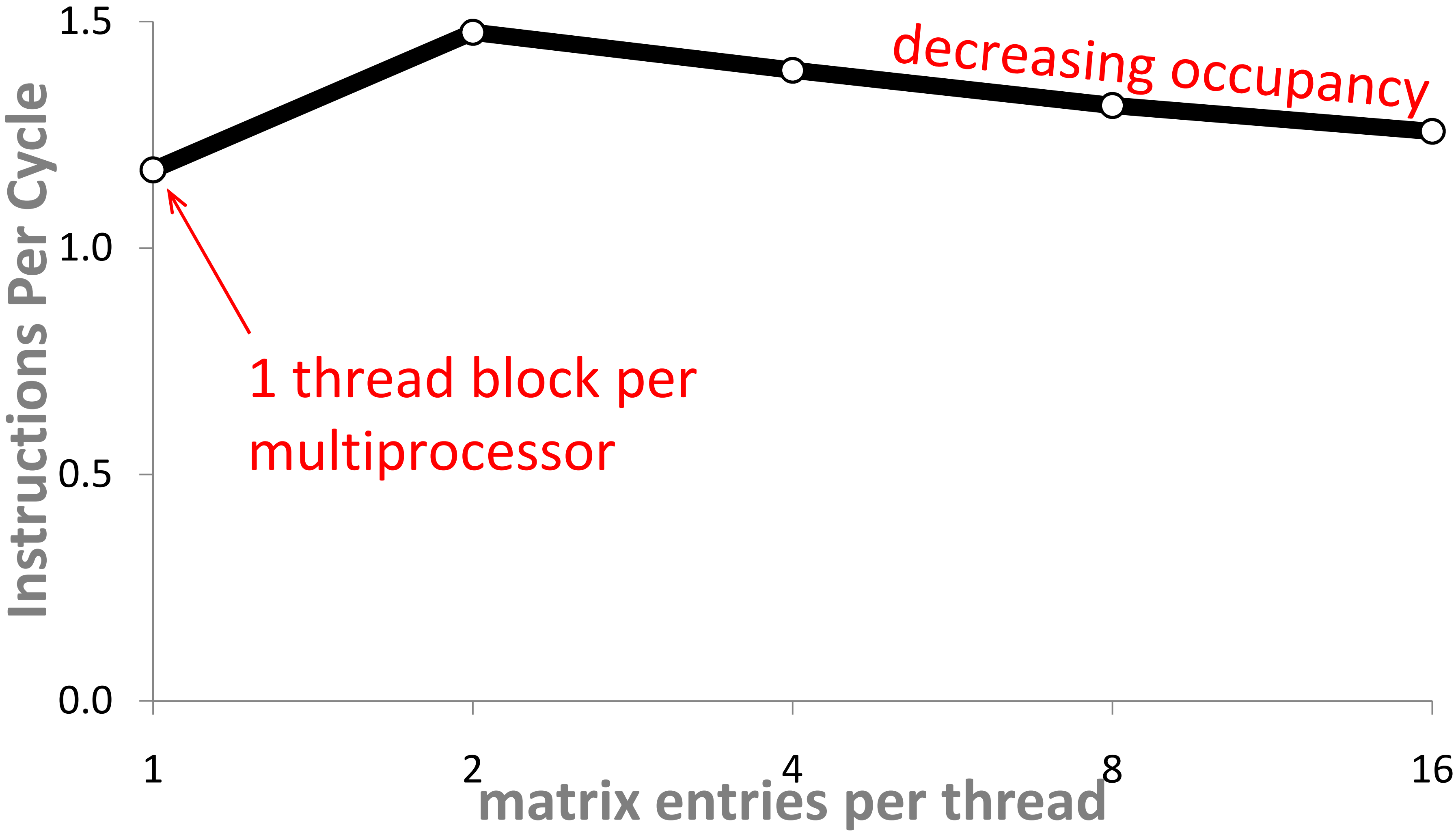
Fewer instructions per each solve

Poor latency hiding w/ 1 thread block per SM

- First thing block does – access global memory
- Can't do any computing until data comes
- So, can't hide latency

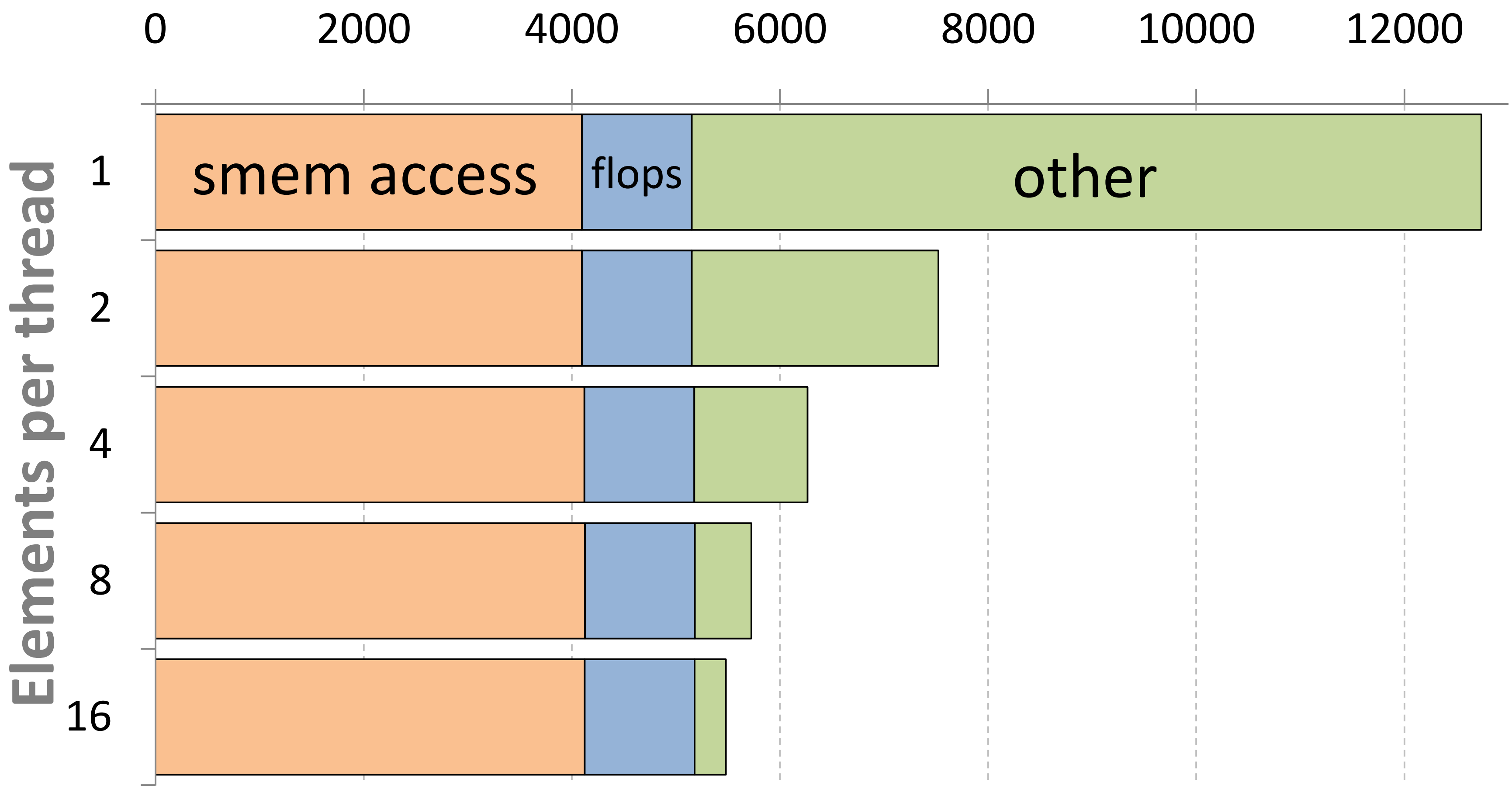
- Need to have at least 2 concurrent blocks
  – Not possible if using 32x32 thread blocks

Instruction throughput



Instruction throughput decreases – but runs faster?

Instructions executed per thread block



Dramatically fewer auxiliary instructions (control, barriers, etc.)

- Similar effect as with classical loop unrolling

Most instructions are shared memory access?!

Why so many shared memory accesses?

How many instructions is this:

```
A[y][x] -= A[y][i]*A[i][x];
```

- 1 arithmetic instruction (FMA)
- 3 loads, 1 store

Note: each load costs 2 arithmetic instructions

- 32 banks vs 32 streaming processors
- But run at half clock rate

These 3 loads are 6x more expensive than 1 FMA

- Eliminate some?

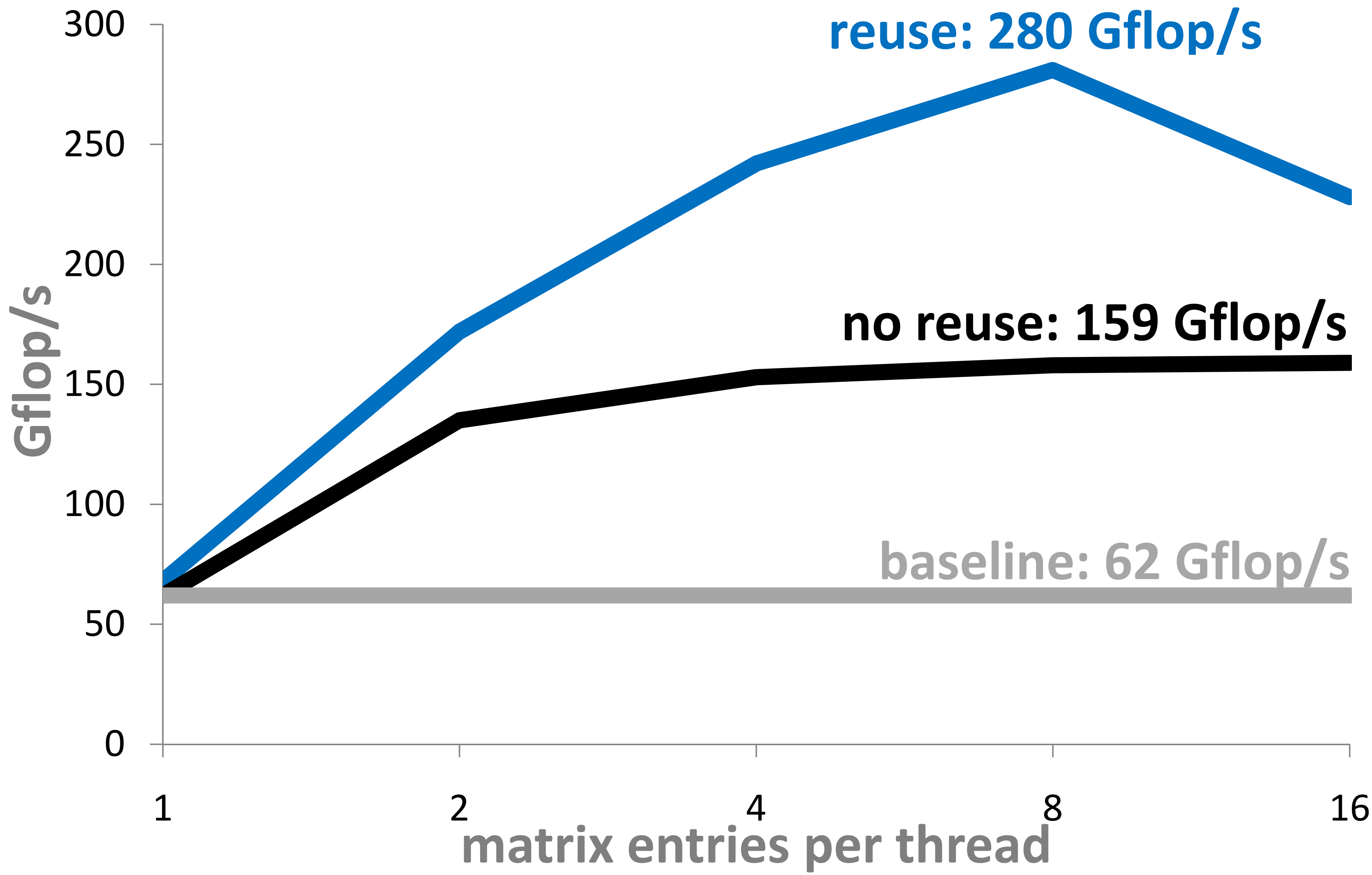
Look for reuse

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    for( int j = 8*y; j < 8*(y+1); j++ )
        A[j][x] = in[32*32*problem+32*j+x];

    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/8 ) A[i][x] /= A[i][i];
        __syncthreads( );
        for( int j = 8*y; j < 8*(y+1); j++ )
            if( j != i ) A[j][x] -= A[j][i]*A[i][x];
    }

    for( int j = 8*y; j < 8*(y+1); j++ )
        out[32*32*problem+32*j+x] = A[j][x];
}
```

Reuse local copies instead

```
__global__ void eliminate( float *in, float *out ) {
    int x = threadIdx.x, y = threadIdx.y, problem = blockIdx.x;

    float a[8]; // array in registers
    for( int j = 0; j < 8; j++ )
        a[j] = A[8*y+j][x] = in[32*32*problem+32*(8*y+j)+x];

    #pragma unroll
    for( int i = 0; i < 32; i++ )
    {
        if( y == i/8 ) A[i][x] = a[i%8] /= A[i][i];
        __syncthreads( );
        float Aix = A[i][x];
        for( int j = 0; j < 8; j++ )
            if( 8*y+j != i ) A[8*y+j][x] = a[j] -= A[8*y+j][i]*Aix;
    }

    for( int j = 0; j < 8; j++ )
        out[32*32*problem+32*(8*y+j)+x] = a[j];
}
```

The effect: further 1.8x speedup



Conclusion

- Simple optimization technique
- Resembles loop unrolling
- Often results in 2x speedup