1. The following analysis is done for a thread, not for the program as a whole.

   (a)
```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
  int pitch_a, int pitch_b, int pitch_c,
  int n, int m, int p )
{
    int i = blockIdx.x*32 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
      sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}
```

   Each thread will perform $2P$ FLOPs and will access global memory $2P+1$ times, giving a ratio of about $\frac{1}{1}$.

   (b)
```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
  int pitch_a, int pitch_b, int pitch_c,
  int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*32 + tx;
    int j = blockIdx.y;
    __shared__ float cb[32];

    float sum = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
      cb[tx] = c[ks+tx+pitch_c*j];
      // Why dont we need a __syncthreads() here ?
      for( int k = ks; k < ks+32; ++k )
        sum += b[i+pitch_b*k] * cb[k-ks];
    }
    a[i+pitch_a*j] = sum;
}
```

   There will be less global memory access due to the introduction of cb. A thread will still perform $2P$ FLOPs. However, a thread will make $\frac{P}{32}$ accesses to global memory when accessing c, P accesses to global memory when accessing b, and one global memory access when accessing a. So the ratio is $\frac{\frac{P}{32}+P+1}{2P} \approx \frac{1}{2}$.

   (c)
```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
  int pitch_a, int pitch_b, int pitch_c,
  int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*64 + tx;
```

```
    int j = blockIdx.y;
    __shared__ float cb[32];

    float sum0 = 0.0, sum1 = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
      cb[tx] = c[ks+tx+pitch_c*j];
      __syncthreads();
      for( int k = ks; k < ks+32; ++k ){
        sum0 += b[i+pitch_b*k] * cb[k-ks];
        sum1 += b[i+32+pitch_b*k] * cb[k-ks];
      }
      __syncthreads();
    }
    a[i+pitch_a*j] = sum0;
    a[i+32+pitch_a*j] = sum1;
}
```

Every thread will perform $4P$ FLOPs . There are 2 global memory accesses due to a, $\frac{P}{32}$ global memory accesses due to c, and $2P$ global memory accesses due to b. The ratio is $\frac{2P+\frac{P}{32}+2}{4P} \approx \frac{1}{2}$. We should expect better performance here than with k2, however, due to the loop unrolling.

(d)
```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
  int pitch_a, int pitch_b, int pitch_c,
  int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*32 + tx;
    int j = blockIdx.y*2;
    __shared__ float cb0[32], cb1[32];

    float sum0 = 0.0, sum1 = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
      cb0[tx] = c[ks+tx+pitch_c*j];
      cb1[tx] = c[ks+tx+pitch_c*(j+1)];
      __syncthreads();
      for( int k = ks; k < ks+32; ++k ){
        float rb = b[i+pitch_b*k];
        sum0 += rb * cb0[k-ks];
        sum1 += rb * cb1[k-ks];
      }
      __syncthreads();
    }
    a[i+pitch_a*j] = sum0;
    a[i+pitch_a*(j+1)] = sum1;
}
```

A thread running this version of the kernel will perform $4P$ FLOPs, $2\frac{P}{32}$ global memory accesses due to c, P

global memory accesses due to b, and 2 global memory accesses due to a. The ratio is $\frac{\frac{P}{16}+P+2}{4P} \approx \frac{1}{4}$.

We expect the runtime of the kernels to be decreasing. If we assume that global memory access is the only expensive transaction occuring during the run of the kernel, then we will expect k1 to have a performance of x GLOPS, and k2 and k3 to both produce $2x$ GLOPS, and then k4 to produce $4x$ GFLOPS. We do not see this performance trend. Evidently, global memory access is not the only consideration to be made when optimizing kernel codes. Note that k4 does not even double the performance of k1.

RESULTS

```
$ ./mmdriver -bin k1.bin -block 32 1024 -thread 32 1 -mat 1024 -size 1024 -check
binfile=k1.bin array=1024x1024 matrix=1024x1024 block=<32x1024> thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 32x1024
 block = 32x1x1
 flops = 2147483648
  msec =      87996   GFLOPS =   24.40,   26.49 (kernel)
no errors found

$ ./mmdriver -bin k2.bin -block 32 1024 -thread 32 1 -mat 1024 -size 1024 -check
binfile=k2.bin array=1024x1024 matrix=1024x1024 block=<32x1024> thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 32x1024
 block = 32x1x1
 flops = 2147483648
  msec =      94413   GFLOPS =   22.75,   24.59 (kernel)
no errors found

$ ./mmdriver -bin k3.bin -block 16 1024 -thread 32 1 -mat 1024 -size 1024 -check
binfile=k3.bin array=1024x1024 matrix=1024x1024 block=<16x1024> thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 16x1024
 block = 32x1x1
 flops = 2147483648
  msec =      53414   GFLOPS =   40.20,   46.06 (kernel)
no errors found

$ ./mmdriver -bin k4.bin -block 32 512 -thread 32 1 -mat 1024 -size 1024 -check
binfile=k4.bin array=1024x1024 matrix=1024x1024 block=<32x512> thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 32x512
 block = 32x1x1
 flops = 2147483648
  msec =      52761   GFLOPS =   40.70,   47.17 (kernel)
no errors found
```

2. The two way unroll on i and j gives amazing performance.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
  int pitch_a, int pitch_b, int pitch_c,
  int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*64 + tx;
    int j = blockIdx.y*2;
    __shared__ float cb0[32], cb1[32];

    float sum0 = 0.0, sum1 = 0., sum2 = 0.0, sum3 = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
      cb0[tx] = c[ks+tx+pitch_c*j];
      cb1[tx] = c[ks+tx+pitch_c*(j+1)];
      __syncthreads();
      for( int k = ks; k < ks+32; ++k ){
        float rb0 = b[i+pitch_b*k];
        float rb1 = b[i+32+pitch_b*k];

        sum0 += rb0 * cb0[k - ks];
        sum1 += rb0 * cb1[k - ks];

        sum2 += rb1*cb0[k - ks];
        sum3 += rb1*cb1[k - ks];

      }
      __syncthreads();
    }
    a[i+pitch_a*j] = sum0;
    a[i+pitch_a*(j+1)] = sum1;
    a[i+32+pitch_a*j] = sum2;
    a[i+32+pitch_a*(j+1)] = sum3;
}
```

RESULTS

```
$ ./mmdriver -bin k5.bin -block 16 512 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k5.bin  array=1024x1024  matrix=1024x1024  block=<16x512>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 16x512
 block = 32x1x1
 flops = 2147483648
  msec =      31723   GFLOPS =   67.69,   86.01 (kernel)
no errors found
```

3. I changed the mmgold() function used in the driver to check for errors. Here is what I changed it to :

```
void
mmgold( float* a, int an, float* b, int bn, float* c, int cn, int n1, int n2, int n3 )
{
    int i, j, k;
    for( j = 0; j < n2; ++j )
        for( k = 0; k < n3; ++k )
            for( i = 0; i < n1; ++i )
                a[j+i*an] += b[i+k*bn]*c[j+k*cn];
}
```

And here is the initial performance:

```
$ ./mmdriver -bin k6.bin -block 32 1024 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<32x1024>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 32x1024
 block = 32x1x1
 flops = 2147483648
  msec =     178320   GFLOPS =   12.04,   12.53 (kernel)
no errors found
```

If you just want to see my best attempt, turn to **Attempt 5**. If you want to see all of thinking I went through on my way to getting this performance, you can look at the previous attempts.

Attempt 1 I tried a number of different optimizations. Here is the code and my results with a 4 way unroll on j. This will give some performance increase because of the loop unrolling. Unfortunately, the global memory accesses are not coalesced for the writes to A.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
        int i = blockIdx.x*32 + threadIdx.x;
        int j = blockIdx.y*4;
        float sum0 = 0.0, sum1 = 0.0, sum2=0.0, sum3=0.0;

        for(int k = 0; k < p; ++k)
        {
                float b_tmp = b[k*pitch_b+i];
                float c_tmp0 = c[j+pitch_c*k];
                float c_tmp1 = c[j+1+pitch_c*k];
                float c_tmp2 = c[j+2+pitch_c*k];
                float c_tmp3 = c[j+3+pitch_c*k];
                sum0 += b_tmp*c_tmp0;
                sum1 += b_tmp*c_tmp1;
                sum2 += b_tmp*c_tmp2;
                sum3 += b_tmp*c_tmp3;
        }
```

```
        a[j+pitch_a*i] = sum0;
        a[j+1+pitch_a*i]=sum1;
        a[j+2+pitch_a*i]=sum2;
        a[j+3+pitch_a*i]=sum3;
}
```

```
$ ./mmdriver -bin k6.bin -block 32 256 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<32x256>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 32x256
 block = 32x1x1
 flops = 2147483648
  msec =      60376   GFLOPS =   35.57,   42.09 (kernel)
no errors found
```

Attempt 2 Here is a second attempt with a 2 way unroll on i and j.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
        int i = blockIdx.x*64 + threadIdx.x;
        int j = blockIdx.y*2;
        float sum0 = 0.0, sum1 = 0.0, sum2=0.0, sum3=0.0;

        for(int k = 0; k < p; ++k)
        {
                float b_tmp = b[k*pitch_b+i];
                float b_tmp2 = b[k*pitch_b+i+32];
                float c_tmp = c[j+pitch_c*k];
                float c_tmp2 = c[j+1+pitch_c*k];
                sum0 += b_tmp*c_tmp;
                sum1 += b_tmp2*c_tmp;
                sum2 += b_tmp*c_tmp2;
                sum3 += b_tmp2*c_tmp2;
        }

        a[j+pitch_a*i] = sum0;
        a[j+pitch_a*(i+32)] = sum1;
        a[j+1+pitch_a*i]=sum2;
        a[j+1+pitch_a*(i+32)] =sum3;
}
```

```
$ ./mmdriver -bin k6.bin -block 16 512 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<16x512>  thread=<32x1>
matrix = 1024x1024
```

```
 array = 1024x1024
  grid = 16x512
 block = 32x1x1
 flops = 2147483648
  msec =      58039   GFLOPS =   37.00,   43.86 (kernel)
no errors found
```

Attempt 3 And here I tried to do a 4 way unroll on i, along with a transformation of the problem to $C^T B = A^T$, along with an index transformation such that A is stored, and not A^T . This allows us to coalese the writes to A in global memory.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
        int i = blockIdx.x*128 + threadIdx.x;
        int j = blockIdx.y;
        float sum0 = 0.0, sum1 =0.0, sum2 = 0.0, sum3 = 0.0;
        for( int k = 0; k < p; ++k )
        {
                float c_tmp0= c[k*pitch_c+i];
                float c_tmp1 = c[k*pitch_c + i + 32];
                float c_tmp2 = c[k*pitch_c + i + 64];
                float c_tmp3 = c[k*pitch_c + i + 96];
                float b_tmp0 = b[j+pitch_b*k];
                sum0 += b_tmp0*c_tmp0;
                sum1 += b_tmp0*c_tmp1;
                sum2 += b_tmp0*c_tmp2;
                sum3 += b_tmp0*c_tmp3;
        }
        a[i+pitch_a*j] = sum0;
        a[i+32+pitch_a*j] = sum1;
        a[i+64+pitch_a*j] = sum2;
        a[i+96+pitch_a*j] = sum3;
}
```

```
$ ./mmdriver -bin k6.bin -block 8 1024 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<8x1024>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 8x1024
 block = 32x1x1
 flops = 2147483648
  msec =      57345   GFLOPS =   37.45,   42.83 (kernel)
no errors found
```

And interestingly enough, when I get rid of the unnecessary local c variables, I get some performance increase, but not enough to be very useful. This is just an observation.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
        int i = blockIdx.x*128 + threadIdx.x;
        int j = blockIdx.y;
        float sum0 = 0.0, sum1 =0.0, sum2 = 0.0, sum3 = 0.0;
        for( int k = 0; k < p; ++k )
        {
                float b_tmp0 = b[j+pitch_b*k];
                sum0 += b_tmp0*c[k*pitch_c+i];
                sum1 += b_tmp0*c[k*pitch_c + i + 32];
                sum2 += b_tmp0*c[k*pitch_c + i + 64];
                sum3 += b_tmp0*c[k*pitch_c + i + 96];
        }
        a[i+pitch_a*j] = sum0;
        a[i+32+pitch_a*j] = sum1;
        a[i+64+pitch_a*j] = sum2;
        a[i+96+pitch_a*j] = sum3;
}
```

```
$ ./mmdriver -bin k6.bin -block 8 1024 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<8x1024>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 8x1024
 block = 32x1x1
 flops = 2147483648
  msec =      56665   GFLOPS =   37.90,   43.50 (kernel)
no errors found
```

I really expected this to solve my problem, because global memory accesses are still coalesed with respect to b and c, and now the access is also coalesced with respect to a! But, I get negligible performance increase by coalescing my access to a. I wonder why this is?

Attempt 4 Here I keep the index transformation to get global memory coalescing for A, I also try a four way unroll on j.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
        int i = blockIdx.x*32 + threadIdx.x;
        int j = blockIdx.y*4;
        float sum0 = 0.0, sum1 =0.0, sum2 = 0.0, sum3 = 0.0;
        for( int k = 0; k < p; ++k )
        {
                float c_tmp = c[k*pitch_c+i];
```

```
                sum0 += b[j+pitch_b*k]*c_tmp;
                sum1 += b[j+1+pitch_b*k]*c_tmp;
                sum2 += b[j+2+pitch_b*k]*c_tmp;
                                sum3 += b[j+3+pitch_b*k]*c_tmp;
        }
        a[i+pitch_a*j] = sum0;
        a[i+pitch_a*(j+1)] = sum1;
        a[i+pitch_a*(j+2)] = sum2;
        a[i+pitch_a*(j+3)] = sum3;
}
```

```
$./mmdriver -bin k6.bin -block 32 256 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<32x256>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 32x256
 block = 32x1x1
 flops = 2147483648
  msec =      57885   GFLOPS =   37.10,   42.33 (kernel)
no errors found
```

Attempt 5 Here is a code with coalesced memory accesses to A(coming from the previously described matrix transformation), along with four way loop unrolling on i and two way loop unrolling on j. This is my best attempt.

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
        int i = blockIdx.x*128 + threadIdx.x;
        int j = blockIdx.y*2;
        float sum0 = 0.0, sum1 =0.0, sum2 = 0.0, sum3 = 0.0;
        for( int k = 0; k < p; ++k )
        {
                float b_tmp0 = b[j+pitch_b*k];
                float b_tmp1 = b[j+1+pitch_b*k];
                sum0 += b_tmp0*c[k*pitch_c+i];
                sum1 += b_tmp0*c[k*pitch_c + i + 32];
                sum2 += b_tmp0*c[k*pitch_c + i + 64];
                sum3 += b_tmp0*c[k*pitch_c + i + 96];

                sum4 += b_tmp1*c[k*pitch_c+i];
                                sum5 += b_tmp1*c[k*pitch_c + i + 32];
                sum6 += b_tmp1*c[k*pitch_c + i + 64];
                sum7 += b_tmp1*c[k*pitch_c + i + 96];
        }
        a[i+pitch_a*j] = sum0;
        a[i+32+pitch_a*j] = sum1;
        a[i+64+pitch_a*j] = sum2;
```

```
        a[i+96+pitch_a*j] = sum3;

        a[i+pitch_a*(j+1)] = sum4;
        a[i+32+pitch_a*(j+1)] = sum5;
        a[i+64+pitch_a*(j+1)] = sum6;
        a[i+96+pitch_a*(j+1)] = sum7;
}
```

```
$./mmdriver -bin k6.bin -block 8 512 -thread  32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<8x512>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 8x512
 block = 32x1x1
 flops = 2147483648
  msec =      33235   GFLOPS =   64.62,   81.49 (kernel)
no errors found
```

Update

We just got a different driver on Carmen. Here I run my code using this driver. I called the executable td.

```
$ ./td -bin k6.bin -block 8 512 -thread 32 1 -mat 1024 -size 1024 -check
binfile=k6.bin  array=1024x1024  matrix=1024x1024  block=<8x512>  thread=<32x1>
matrix = 1024x1024
 array = 1024x1024
  grid = 8x512
 block = 32x1x1
 flops = 2147483648
  msec =      41167   GFLOPS =   52.17,   81.24 (kernel)
no errors found
```