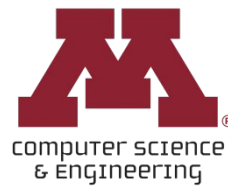


CSCI 5451: Introduction to Parallel Computing

Lecture 23: cuBLAS GEMM replication (cont'd)



Original worklog [here](#)



UNIVERSITY OF MINNESOTA
Driven to Discover®

Announcements (11/24)

- ❑ Be sure to set your project meeting [here](#). If you do not meet with me before this Thursday with your group, you will receive 0/4 points.
- ❑ As a reminder HW3 is out & due Nov 28
- ❑ HW4 out (Convolution operation on CUDA)
- ❑ Wednesday Lecture *Cancelled* this week
- ❑ Block-Cyclic Decomposition Visualizations [here](#)



Overview

- ❑ GEMM background
- ❑ Worklog
 - Naive Kernel
 - Coalescing
 - Shared Memory
 - Thread Coarsening
 - Vectorized Memory Loading
 - Autotuning



Overview

❑ **GEMM background**

❑ Worklog

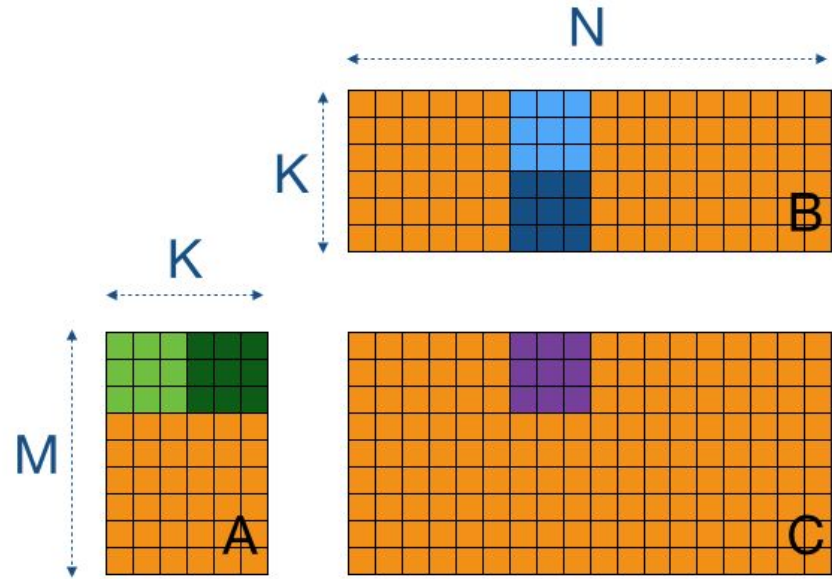
- Naive Kernel
- Coalescing
- Shared Memory
- Thread Coarsening
- Vectorized Memory Loading
- Autotuning



GEMM Background

Given matrices A , B , C and floating point values α , β , GEMM involves computing the following:

$$C = \alpha * (A * B) + \beta * C$$



The diagram shows a 3x3 purple grid (representing a block in C) followed by an equals sign, then a 3x3 green grid (representing a block in A) followed by an 'x' (multiplication), then a 3x3 blue grid (representing a block in B), followed by a plus sign, then a 3x3 dark green grid (representing another block in A) followed by an 'x' (multiplication), then a 3x3 dark blue grid (representing another block in B).

GEMM Background

We'll be walking through a kernel replication of GEMM that aims to improve performance gradually by introducing concepts we have discussed in the previous lectures (plus a new concept)

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



GEMM Background

We'll be walking through a kernel replication of GEMM that aims to improve performance gradually by introducing concepts we have discussed in the previous lectures (plus a new concept)

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



Overview

- ❑ GEMM background
- ❑ Worklog
 - **Naive Kernel**
 - Coalescing
 - Shared Memory
 - Thread Coarsening
 - Vectorized Memory Loading
 - Autotuning



Naive Kernel

```
// create as many blocks as necessary to map all of C
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
// 32 * 32 = 1024 thread per block
dim3 blockDim(32, 32, 1);
// launch the asynchronous execution of the kernel on the device
// The function call returns immediately on the host
sgemm_naive<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

```
_global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                          const float *B, float beta, float *C) {
    // compute position in C that this thread is responsible for
    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
    const uint y = blockIdx.y * blockDim.y + threadIdx.y;

    // `if` condition is necessary for when M or N aren't multiples of 32.
    if (x < M && y < N) {
        float tmp = 0.0;
        for (int i = 0; i < K; ++i) {
            tmp += A[x * K + i] * B[i * N + y];
        }
        //  $C = \alpha(A @ B) + \beta * C$ 
        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
    }
}
```



Naive Kernel

- ❑ For all the following tests the GEMM kernel is performed on matrices of 4092x4092 on an A6000 GPU
- ❑ The current requirements are
 - Operations
 - ✓ $2 * 4092^3 + 4092^2 = 137\text{GFLOPS}$
 - Read data
 - ✓ $3 * 4092^2 * 4\text{B} = 201\text{MB}$
 - Write data
 - ✓ $4092^2 * 4\text{B} = 67\text{MB}$



Naive Kernel

- ❑ For all the following tests the GEMM kernel is performed on matrices of 4092x4092 on an A6000 GPU
- ❑ The current requirements are
 - Operations
 - ✓ $2 * 4092^3 + 4092^2 = 137\text{GFLOPS}$
 - Read data
 - ✓ $3 * 4092^2 * 4\text{B} = 201\text{MB}$
 - Write data
 - ✓ $4092^2 * 4\text{B} = 67\text{MB}$

A6000 Specs:

- 30 TFLOPS/Second
- 768GB/s

Theoretical “Best”:

- 4.5ms for compute
- .34s memory transfers

Current FLOPS:

~300GFLOPS (500ms)



Overview

- ❑ GEMM background
- ❑ Worklog
 - Naive Kernel
 - **Coalescing**
 - Shared Memory
 - Thread Coarsening
 - Vectorized Memory Loading
 - Autotuning



Memory Coalescing

How can we use memory coalescing to improve this kernel?

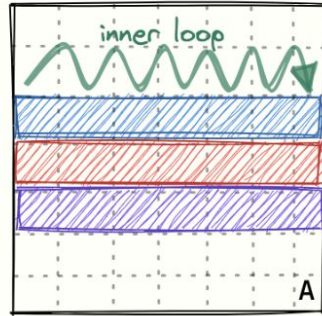
```
_global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                           const float *B, float beta, float *C) {
    // compute position in C that this thread is responsible for
    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
    const uint y = blockIdx.y * blockDim.y + threadIdx.y;

    // `if` condition is necessary for when M or N aren't multiples of 32.
    if (x < M && y < N) {
        float tmp = 0.0;
        for (int i = 0; i < K; ++i) {
            tmp += A[x * K + i] * B[i * N + y];
        }
        //  $C = \alpha * (A @ B) + \beta * C$ 
        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
    }
}
```



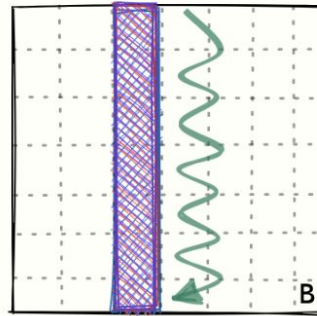
Memory Coalescing

Naive kernel:



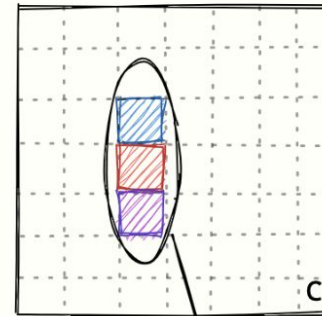
threads access non-consecutive
values \Rightarrow cannot coalesce

@



all threads access same
values \Rightarrow within-warp broadcast

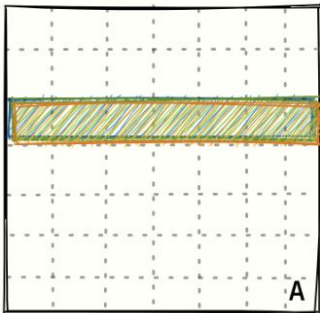
=



No benefit to
putting these threads
in same warp

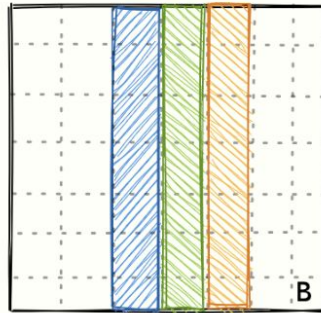
Memory Coalescing

Coalescing kernel:



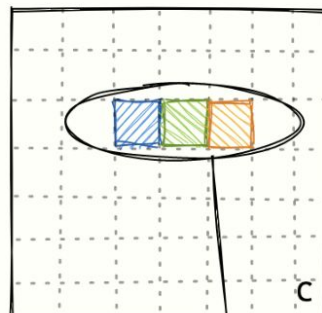
all threads access same values \Rightarrow within-warp broadcast

@



threads access consecutive values \Rightarrow can coalesce

"



Make sure these threads end up in same warp to exploit coalescing

Memory Coalescing

```
// gridDim stays the same  
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));  
// make blockDim 1-dimensional, but don't change number of threads  
dim3 blockDim(32 * 32);  
sgemm_coalescing<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

Relevant
portion of
program

```
const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);  
const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);  
  
if (x < M && y < N) {  
    float tmp = 0.0;  
    for (int i = 0; i < K; ++i) {  
        tmp += A[x * K + i] * B[i * N + y];  
    }  
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];  
}
```



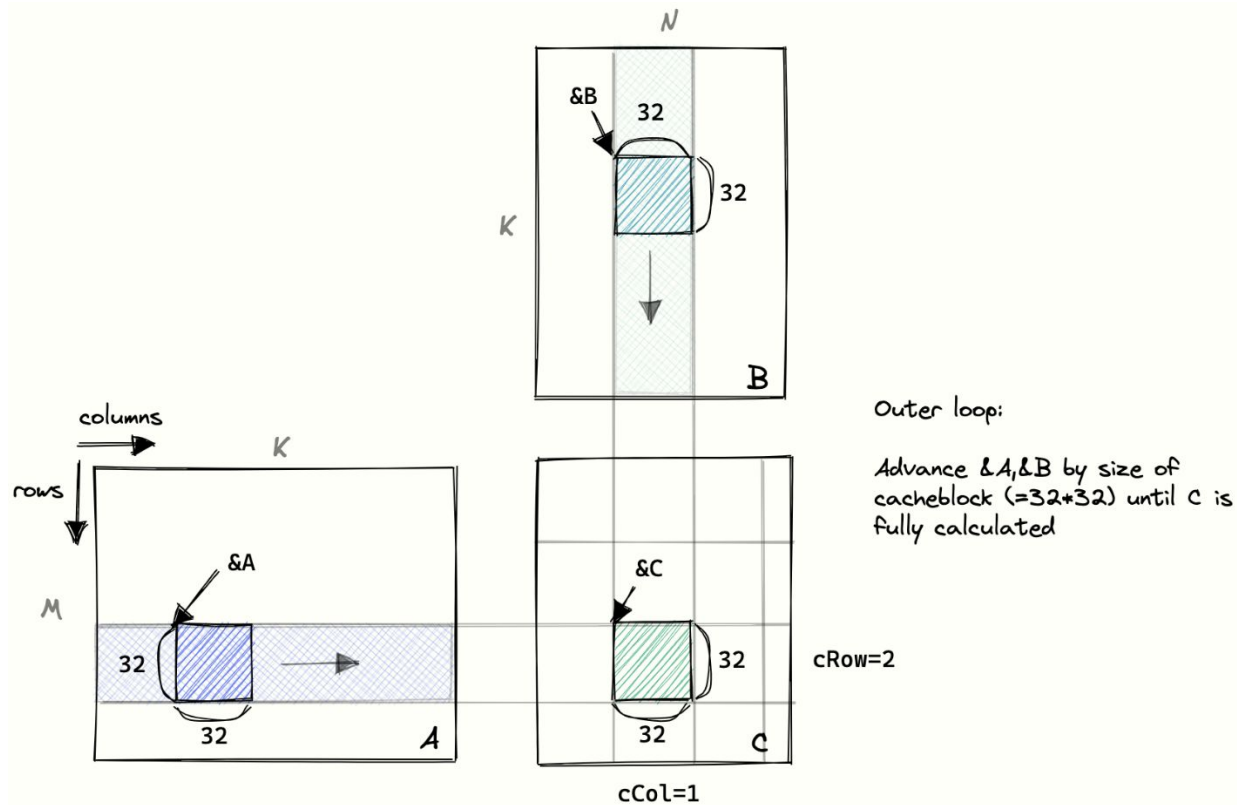
Memory Coalescing

Updated Kernel throughput: ~2000 GFLOPS

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



Shared Memory



Overview

- ❑ GEMM background
- ❑ Worklog
 - Naive Kernel
 - Coalescing
 - **Shared Memory**
 - Thread Coarsening
 - Vectorized Memory Loading
 - Autotuning



Shared Memory

```
__global__ void sgemm_shared_mem_block(int M, int N, int K, float alpha,
                                       const float *A, const float *B,
                                       float beta, float *C) {

    const int BLOCKSIZE=32;
    // the output block that we want to compute in this threadblock
    const uint cRow = blockIdx.x;
    const uint cCol = blockIdx.y;

    // allocate buffer for current block in fast shared mem
    // shared mem is shared between all threads in a block
    __shared__ float As[BLOCKSIZE * BLOCKSIZE];
    __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

    // the inner row & col that we're accessing in this thread
    const uint threadCol = threadIdx.x % BLOCKSIZE;
    const uint threadRow = threadIdx.x / BLOCKSIZE;

    // advance pointers to the starting positions
    A += cRow * BLOCKSIZE * K;           // row=cRow, col=0
    B += cCol * BLOCKSIZE;               // row=0, col=cCol
    C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol
```



```

float tmp = 0.0;
for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    // Have each thread load one of the elements in A & B
    // Make the threadCol (=threadIdx.x) the consecutive index
    // to allow global memory access coalescing
    As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];

    // block threads in this block until cache is fully populated
    __syncthreads();
    A += BLOCKSIZE;
    B += BLOCKSIZE * N;

    // execute the dotproduct on the currently cached block
    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
        tmp += As[threadRow * BLOCKSIZE + dotIdx] *
            Bs[dotIdx * BLOCKSIZE + threadCol];
    }
    // need to sync again at the end, to avoid faster threads
    // fetching the next block into the cache before slower threads are done
    __syncthreads();
}
C[threadRow * N + threadCol] =
    alpha * tmp + beta * C[threadRow * N + threadCol];
}

```



Shared Memory

Updated Kernel throughput: ~3000 GFLOPS

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



Overview

- ❑ GEMM background
- ❑ Worklog
 - Naive Kernel
 - Coalescing
 - Shared Memory
 - **Thread Coarsening**
 - Vectorized Memory Loading
 - Autotuning

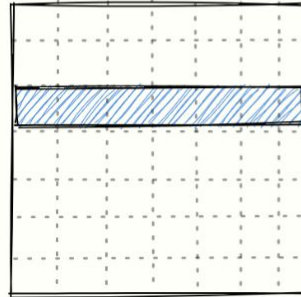


Thread Coarsening

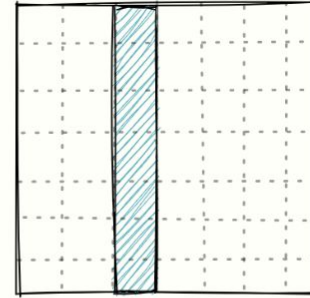
Calculating 1 result per thread requires:

- 7 loads from A
- 7 loads from B
- 1 load & 1 store to C

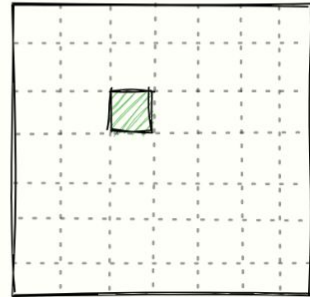
⇒ 15 loads & 1 store per result



A



B



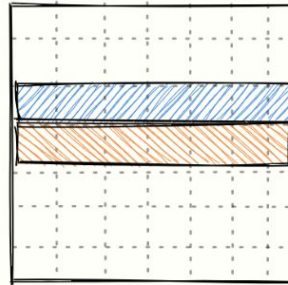
C

Thread Coarsening

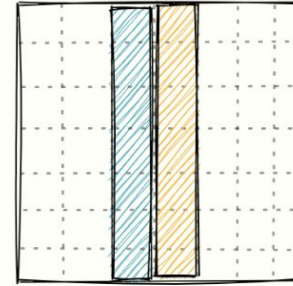
Calculating 4 results per thread requires:

- 14 loads from A
- 14 loads from B
- 4 loads & 4 stores to C

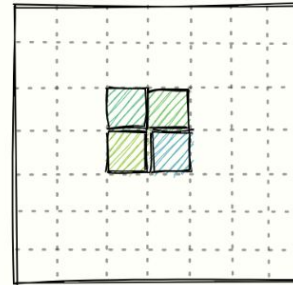
⇒ 8 loads & 1 store per result



A

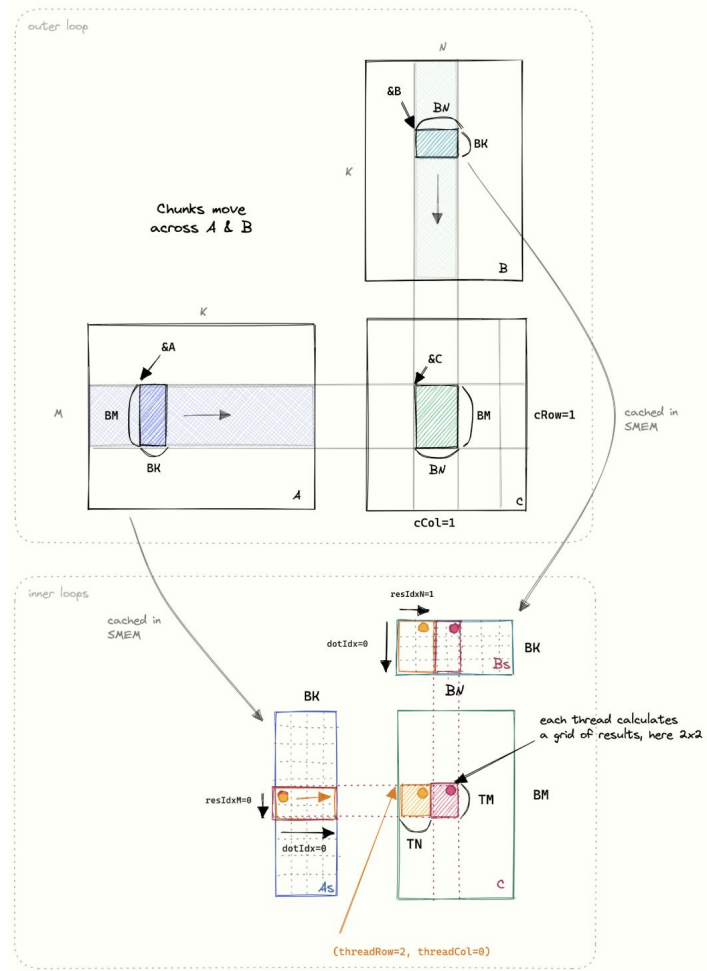


B

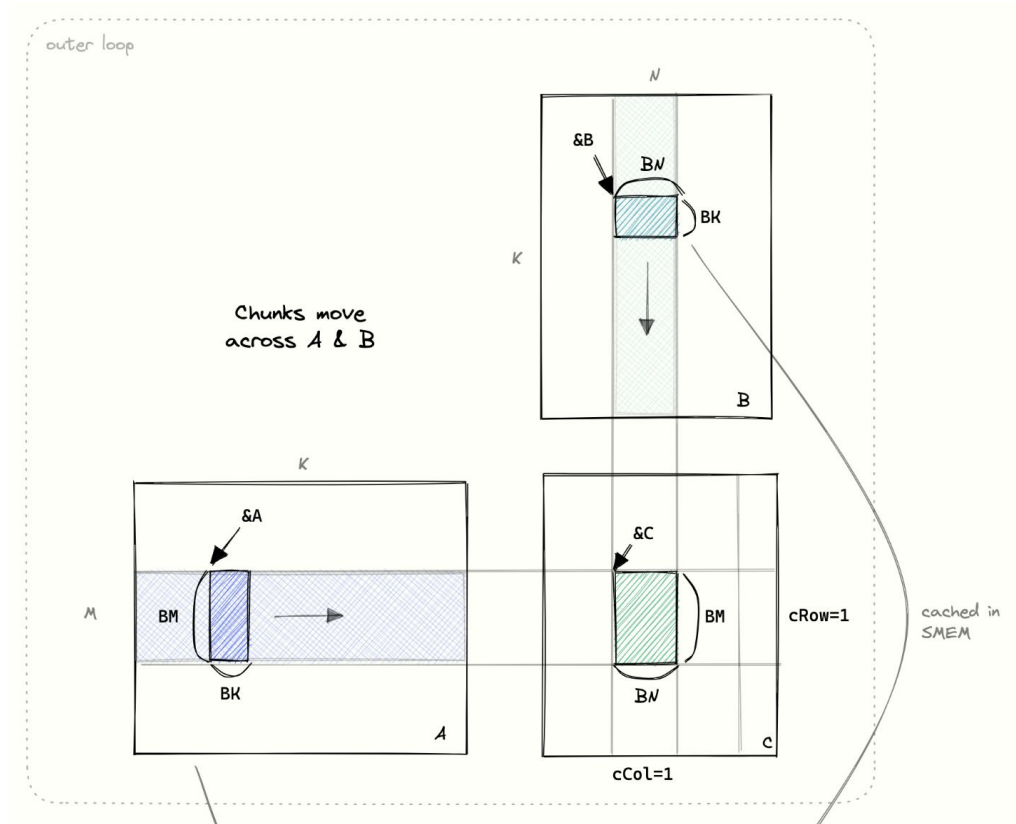


C

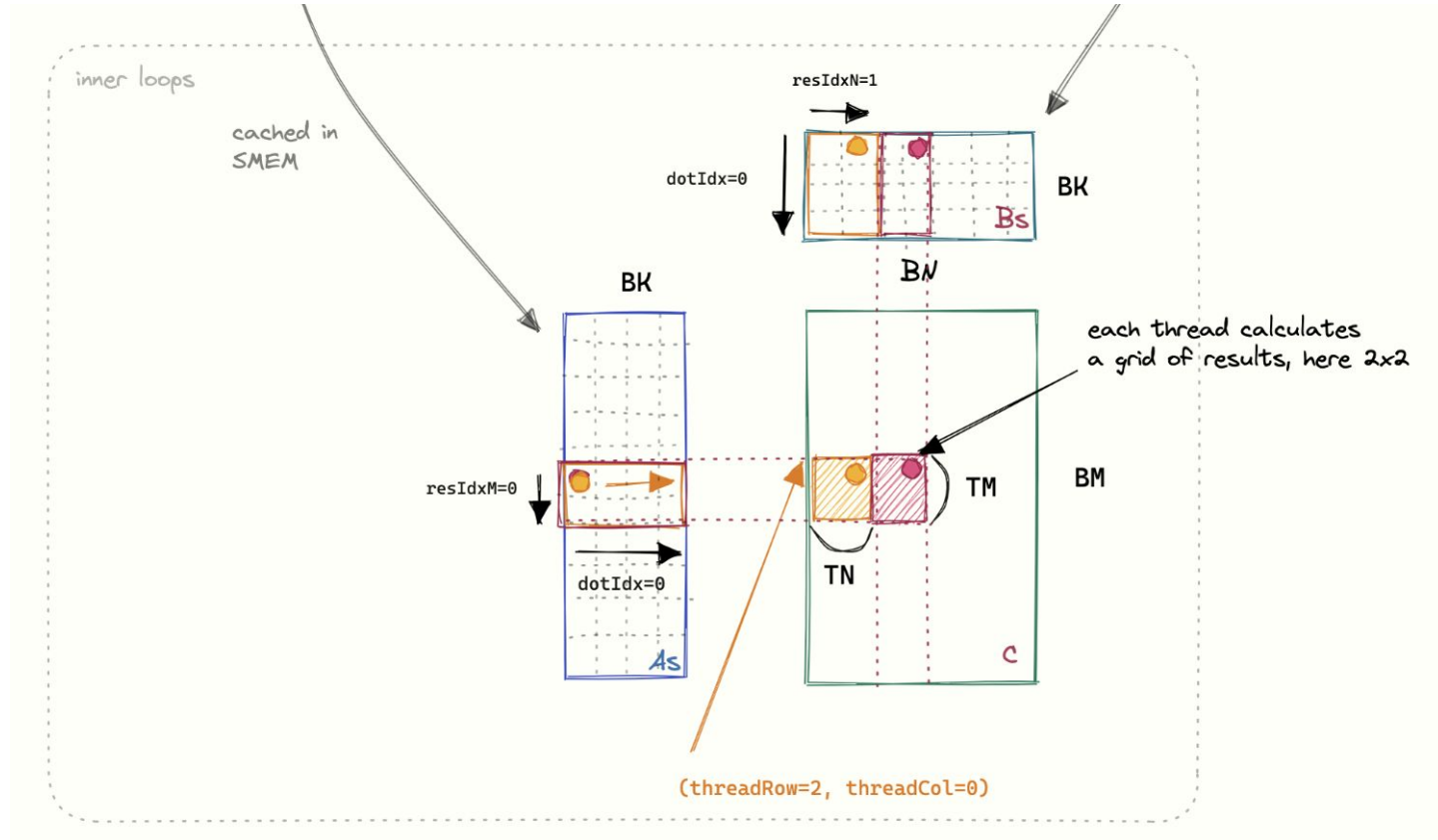
Thread Coarsening



Thread Coarsening



Thread Coarsening



```

__global__ void sgemm2DWarpTiling(...) {
    const int BM = 128;
    const int BN = 128;
    const int BK = 8;
    const int TM = 8;
    const int TN = 8;
    const uint cRow = blockIdx.y;
    const uint cCol = blockIdx.x;

    const uint totalResultsBlocktile = BM * BN;
    // A thread is responsible for calculating TM*TN elements in the blocktile
    const uint numThreadsBlocktile = totalResultsBlocktile / (TM * TN);

    // ResultsPerBlock / ResultsPerThread == ThreadsPerBlock
    assert(numThreadsBlocktile == blockDim.x);

    // BN/TN are the number of threads to span a column
    const int threadCol = threadIdx.x % (BN / TN);
    const int threadRow = threadIdx.x / (BN / TN);

    // allocate space for the current blocktile in smem
    __shared__ float As[BM * BK];
    __shared__ float Bs[BK * BN];

```



```

A += cRow * BM * K;
B += cCol * BN;
C += cRow * BM * N + cCol * BN;

// calculating the indices that this thread will load into SMEM
const uint innerRowA = threadIdx.x / BK;
const uint innerColA = threadIdx.x % BK;
// calculates the number of rows of As that are being loaded in a single step
// by a single block
const uint strideA = numThreadsBlocktile / BK;
const uint innerRowB = threadIdx.x / BN;
const uint innerColB = threadIdx.x % BN;
// for both As and Bs we want each load to span the full column-width, for
// better GMEM coalescing (as opposed to spanning full row-width and iterating
// across columns)
const uint strideB = numThreadsBlocktile / BN;

    // allocate thread-local cache for results in registerfile
float threadResults[TM * TN] = {0.0};
// register caches for As and Bs
float regM[TM] = {0.0};
float regN[TN] = {0.0};

```



```

// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches
    for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) {
        As[(innerRowA + loadOffset) * BK + innerColA] =
            A[(innerRowA + loadOffset) * K + innerColA];
    }
    for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) {
        Bs[(innerRowB + loadOffset) * BN + innerColB] =
            B[(innerRowB + loadOffset) * N + innerColB];
    }
    __syncthreads();

    // advance blocktile
    A += BK;          // move BK columns to right
    B += BK * N;      // move BK rows down

```



```

// calculate per-thread results
for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // block into registers
    for (uint i = 0; i < TM; ++i) {
        regM[i] = As[(threadRow * TM + i) * BK + dotIdx];
    }
    for (uint i = 0; i < TN; ++i) {
        regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
    }
    for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
        for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
            threadResults[resIdxM * TN + resIdxN] +=
                regM[resIdxM] * regN[resIdxN];
        }
    }
}
__syncthreads();
}

```




```
// write out the results
for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
    for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
        C[(threadRow * TM + resIdxM) * N + threadCol * TN + resIdxN] =
            alpha * threadResults[resIdxM * TN + resIdxN] +
            beta * C[(threadRow * TM + resIdxM) * N + threadCol * TN + resIdxN];
    }
}
}
```



Thread Coarsening

Updated Kernel throughput: ~16000 GFLOPS

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



Overview

- ❑ GEMM background
- ❑ Worklog
 - Naive Kernel
 - Coalescing
 - Shared Memory
 - Thread Coarsening
 - **Vectorized Memory Loading**
 - Autotuning



Vectorized Access

```
__global__ void sgemmVectorize(...) {  
    const int BM = 128;  
    const int BN = 128;  
    const int BK = 8;  
    const int TM = 8;  
    const int TN = 8;  
  
    const uint cRow = blockIdx.y;  
    const uint cCol = blockIdx.x;  
  
    const uint totalResultsBlocktile = BM * BN;  
    // A thread is responsible for calculating TM*TN elements in the blocktile  
    const uint numThreadsBlocktile = totalResultsBlocktile / (TM * TN);  
  
    // ResultsPerBlock / ResultsPerThread == ThreadsPerBlock  
    assert(numThreadsBlocktile == blockDim.x);  
  
    // BN/TN are the number of threads to span a column  
    const int threadCol = threadIdx.x % (BN / TN);  
    const int threadRow = threadIdx.x / (BN / TN);
```



Vectorized Access

```
// allocate space for the current blocktile in smem
__shared__ float As[BM * BK];
__shared__ float Bs[BK * BN];

// Move blocktile to beginning of A's row and B's column
A += cRow * BM * K;
B += cCol * BN;
C += cRow * BM * N + cCol * BN;

// calculating the indices that this thread will load into SMEM
// we'll load 128bit / 32bit = 4 elements per thread at each step
const uint innerRowA = threadIdx.x / (BK / 4);
const uint innerColA = threadIdx.x % (BK / 4);
// calculates the number of rows of As that are being loaded in a single step
// by a single block
const uint rowStrideA = (numThreadsBlocktile * 4) / BK;
const uint innerRowB = threadIdx.x / (BN / 4);
const uint innerColB = threadIdx.x % (BN / 4);
```



Vectorized Access

```
// for both As and Bs we want each load to span the full column-width, for
// better GMEM coalescing (as opposed to spanning full row-width and iterating
// across columns)
const uint rowStrideB = numThreadsBlocktile / (BN / 4);

// allocate thread-local cache for results in registerfile
float threadResults[TM * TN] = {0.0};
float regM[TM] = {0.0};
float regN[TN] = {0.0};
```



Vectorized Access

```
// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches
    // transpose A while loading it
    float4 tmp =
        reinterpret_cast<float4 *>(&A[innerRowA * K + innerColA * 4])[0];
    As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
    As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
    As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
    As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;

    reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
        reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
    __syncthreads();

    // advance blocktile
    A += BK;          // move BK columns to right
    B += BK * N;      // move BK rows down
```



Vectorized Access

```
// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches
    // transpose A while loading it
    float4 tmp =
        reinterpret_cast<float4 *>(&A[innerRowA * K + innerColA * 4])[0];
    As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
    As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
    As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
    As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;

    reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =
        reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
    __syncthreads();

    // advance blocktile
    A += BK;          // move BK columns to right
    B += BK * N;      // move BK rows down
```



Vectorized Access

```
// calculate per-thread results
for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // block into registers
    for (uint i = 0; i < TM; ++i) {
        regM[i] = As[dotIdx * BM + threadRow * TM + i];
    }
    for (uint i = 0; i < TN; ++i) {
        regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
    }
    for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
        for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
            threadResults[resIdxM * TN + resIdxN] +=
                regM[resIdxM] * regN[resIdxN];
        }
    }
}
__syncthreads();
}
```



Vectorized Access

```
// write out the results
for (uint resIdxM = 0; resIdxM < TM; resIdxM += 1) {
    for (uint resIdxN = 0; resIdxN < TN; resIdxN += 4) {
        // load C vector into registers
        float4 tmp = reinterpret_cast<float4 *>(
            &C[(threadRow * TM + resIdxM) * N + threadCol * TN + resIdxN])[0];
        // perform GEMM update in reg
        tmp.x = alpha * threadResults[resIdxM * TN + resIdxN] + beta * tmp.x;
        tmp.y = alpha * threadResults[resIdxM * TN + resIdxN + 1] + beta * tmp.y;
        tmp.z = alpha * threadResults[resIdxM * TN + resIdxN + 2] + beta * tmp.z;
        tmp.w = alpha * threadResults[resIdxM * TN + resIdxN + 3] + beta * tmp.w;
        // write back
        reinterpret_cast<float4 *>(
            &C[(threadRow * TM + resIdxM) * N + threadCol * TN + resIdxN])[0] =
            tmp;
    }
}
```



Vectorized Access

Updated Kernel throughput: ~18000 GFLOPS

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



Overview

- ❑ GEMM background
- ❑ Worklog
 - Naive Kernel
 - Coalescing
 - Shared Memory
 - Thread Coarsening
 - Vectorized Memory Loading
 - **Autotuning**



Autotuning

```
__global__ void sgemmVectorize(...) {  
    const int BM = 128;  
    const int BN = 128;  
    const int BK = 8;  
    const int TM = 8;  
    const int TN = 8;
```

Vary the coarsening
factor (8x8) and the
block dimensions.

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%



Conclusions

- ❑ This is the tip of the iceberg - there are many more optimizations we can perform
- ❑ The best kernels are handwritten in SASS - which is assembly associated with each individual GPU
 - These kernels (for problems with sufficient theoretical arithmetic intensity) typically get 90-98% of theoretical speedups

