"بسمه تعالی"

تمرین تحویلی ششم برنامه‌نویسی چندهسته‌ای – زهرا لطیفی – ۹۹۲۳۰۶۹

گام صفر

برای پیاده‌سازی این گام، کرنل را عینا مطابق آنچه خواسته شده بود پیاده‌سازی کردیم:

```
__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

    // step 0
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int 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 = α*(A@B)+β*C
        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
    }
}
```

در تابع main هم پس از تعیین سایز ماتریس‌ها و با فرض ضریب آلفا برابر ۱ و بتا برابر ۰، تخصیص حافظه در سمت host را انجام داده و سپس با اعداد رندومی هر ۳ ماتریس را مقداردهی کردیم.

```
int main() {
    const int M = 2048, N = 2048, K = 2048;
    const float alpha = 1.0f, beta = 0.0f;

    // Allocate memory on the host
    float *h_A, *h_B, *h_C;
    h_A = (float *)malloc(M * K * sizeof(float));
    h_B = (float *)malloc(K * N * sizeof(float));
    h_C = (float *)malloc(M * N * sizeof(float));

    // Initialize input matrices
    for (int i = 0; i < M * K; i++) {
        h_A[i] = (float)rand() / RAND_MAX;
    }
    for (int i = 0; i < K * N; i++) {
        h_B[i] = (float)rand() / RAND_MAX;
    }
```

```
    for (int i = 0; i < M * N; i++) {
        h_C[i] = (float)rand() / RAND_MAX;
    }
```

سپس تخصیص حافظه را در سمت device انجام داده و ماتریس‌ها را از host به device منتقل کردیم.

```
for (int i = 0; i < 5; i++) {
        // Allocate memory on the device
        float *d_A, *d_B, *d_C;
        start_time = clock();
        cudaMalloc(&d_A, M * K * sizeof(float));
        cudaMalloc(&d_B, K * N * sizeof(float));
        cudaMalloc(&d_C, M * N * sizeof(float));

        // Copy input matrices to the device
        cudaMemcpy(d_A, h_A, M * K * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, K * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_C, h_C, M * N * sizeof(float), cudaMemcpyHostToDevice);
```

فراخوانی کرنل را به همان شکل که خواسته شده بود انجام داده و پس از انتقال نتیجه به device، حافظه سمت device را آزاد کردیم.

```
        // Launch the 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, d_A, d_B, beta,
d_C);
        cudaDeviceSynchronize();

        // Copy the result back to the host
        cudaMemcpy(h_A, d_A, M * K * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(h_B, d_B, K * N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(h_C, d_C, M * N * sizeof(float), cudaMemcpyDeviceToHost);

        // Free memory on the device
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);

        elapsed_time = clock() - start_time;
        total_time += elapsed_time;
    }
```

این فرآیند را ۵ بار تکرار کرده و از زمان طی شده میانگین گرفتیم و نهایتا حافظه سمت host را هم آزاد کردیم.

```
        printf("Averaged Elapsed Time : %.20f", total_time / (5 *
CLOCKS_PER_SEC));

    // print Results
        //for (int j = 0; j < K * M; j++)
        //{
        //  printf("%f \t", h_A[j]);
        //}

        //for (int j = 0; j < K * N; j++)
        //{
        //  printf("%f \t", h_B[j]);
        //}

        //for (int j = 0; j < M * N; j++)
        //{
        //  printf("%f \t", h_C[j]);
        //}

    // Free memory on the host
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

مدت زمان لازم برای این عملیات با ابعاد ۲۰۴۸ در ۲۰۴۸ و با احتساب malloc و انتقال بین host و device برابر بود با:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time : 1.66019999999999990000
```

و بدون احتساب سایر مراحل و تنها زمان انجام محاسبات برروی GPU:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time : 1.49940000000000010000
```

گام یک

در هر گام همه چیز همانند گام صفر است و تنها فراخوانی کرنل و تعریف آن تغییر می‌کند.

همانطور که در صورت سوال بیان شده، دسترسی به حافظه در گام صفر منجر به عملکرد بسیار پایین می‌شود. از آنجایی که در حال تجزیه خروجی هستیم، اگر سه نخ متوالی را در یک ریسمان مجسم کنیم، فقط در متغیرهای x خود متفاوت هستند، بنابراین آنها سه ردیف را در یک ستون از ماتریس نتیجه محاسبه می‌کنند. در این گام ترتیب محاسبات خود را طوری تغییر می‌دهیم که هر ریسمان یک ردیف از ماتریس نتیجه را محاسبه کند، به این ترتیب می‌توانیم الگوهای دسترسی به حافظه بسیار بهتری داشته باشیم. فقط نحوه محاسبه متغیرهای x و y را تغییر می‌دهیم:

```
__global__ void sgemm_global_mem_coalesce(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

    // step 1
    const int x = blockIdx.x * BLOCK_SIZE + (threadIdx.x / BLOCK_SIZE);
    const int y = blockIdx.y * BLOCK_SIZE + (threadIdx.x % BLOCK_SIZE);

    // `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 = α*(A@B)+β*C
        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
    }
}
```

blockIdx.x و blockIdx.y نمایانگر اندیس‌های بلوک در grid هستند. threadIdx.x اندیس نخ درون بلوک است.

عبارت (blockIdx.x * BLOCK_SIZE + (threadIdx.x / BLOCK_SIZE موقعیت x را در خروجی محاسبه می‌کند. عبارت (blockIdx.y * BLOCK_SIZE + (threadIdx.x % BLOCK_SIZE موقعیت y را در خروجی محاسبه می‌کند. blockIdx.x * BLOCK_SIZE بلوک را به ردیف مربوطه در ماتریس خروجی نگاشت می‌کند. (/ threadIdx.x BLOCK_SIZE) شماره ستون را در آن سطر محاسبه می‌کند. blockIdx.y * BLOCK_SIZE بلوک را به ستون مربوطه در ماتریس خروجی نگاشت می‌کند. (threadIdx.x % BLOCK_SIZE) شاخص ردیف را در آن ستون محاسبه می‌کند.

فراخوانی کرنل هم به صورت زیر است:

```
// Launch the 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_global_mem_coalesce<<<gridDim, blockDim>>>(M, N, K, alpha, d_A,
d_B, beta, d_C);
```

مدت زمان لازم برای این عملیات با احتساب memoryAllocationها و انتقال‌های بین host و device برابر بود با:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time : 0.32540000000000002000
```

و بدون احتساب سایر مراحل و تنها زمان انجام محاسبات برروی GPU:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time : 0.19860000000000000000
```

گام دوم

تا اینجا فقط از حافظه global استفاده کردیم. در حالی که می‌دانیم حافظه مشترکی نیز داریم که نسبت به حافظه global سریع‌تر است. یک بلوک ۳۲ در ۳۲ از حافظه مشترک برای ماتریس‌های A و B اختصاص می‌دهیم و بلوک نتیجه را با استفاده از حافظه مشترک برای هر بلوک محاسبه می‌کنیم.

```
__global__ void sgemm_shared_mem_block(int M, int N, int K, float alpha, const
float* A,
    const float* B, float beta, float* C) {
    extern __shared__ float As[BLOCKSIZE * BLOCKSIZE];
    extern __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

    // Compute the row and column indices for the current thread
    const uint threadCol = threadIdx.x % BLOCKSIZE;
    const uint threadRow = threadIdx.x / BLOCKSIZE;

    // Compute the block row and column indices
    const int cRow = blockIdx.x;
    const int cCol = blockIdx.y;

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

    float tmp = 0.0;
    // The outer loop advances A along the columns and B along the rows
    // until we have fully calculated the result in C.
    for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCK_SIZE) {
        // Have each thread load one of the elements in A & B from
        // global memory into shared memory.
        // Make the threadCol (=threadIdx.x) the consecutive index
        // to allow global memory access coalescing
        As[threadRow * BLOCK_SIZE + threadCol] = A[threadRow * K + threadCol];
        Bs[threadRow * BLOCK_SIZE + threadCol] = B[threadRow * N + threadCol];

        // Block threads in this block until cache is fully populated
        __syncthreads();

        // Advance pointers onto the next chunk
        A += BLOCK_SIZE;
        B += BLOCK_SIZE * N;

        // Execute the dotproduct on the currently cached block
        for (int dotIdx = 0; dotIdx < BLOCK_SIZE; ++dotIdx) {
            tmp += As[threadRow * BLOCK_SIZE + dotIdx] *
                Bs[dotIdx * BLOCK_SIZE + 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];
}
```

عملکرد به این صورت است که دو آرایه حافظه مشترک As و Bs را تعریف می‌کنیم. هر آرایه دارای اندازه BLOCKSIZE * BLOCKSIZE است (با فرض اینکه BLOCKSIZE برابر ۳۲ است.) این آرایه‌ها برای کش کردن بخش‌هایی از ماتریس‌های A و B در حافظه مشترک استفاده می‌شوند.

threadCol اندیس ستون درون بلوک است. threadRow اندیس ردیف درون بلوک است و اندیس‌های بلوک ('blockIdx.x' و 'blockIdx.y') نشان دهنده سطر و ستون ماتریس خروجی C است که این بلوک از نخ‌ها مسئول محاسبه آن است.

پوینترهای A و B و C در موقعیت شروع این بلوک تنظیم می‌شوند: پوینتر A به ابتدای ردیف در ماتریس A مربوط به بلوک فعلی اشاره می‌کند. همینطور پوینتر B و C.

حلقه بیرونی روی تکه‌هایی به اندازه «BLOCK_SIZE» در امتداد بعد مشترک «K» طی می‌شود. در هر تکه، هر نخ عناصر A و B را در حافظه مشترک بارگذاری می‌کند. (As و Bs)

پس از بارگذاری داده‌ها در حافظه مشترک، نخ‌ها با استفاده از __syncthreads همگام می‌شوند. سپس، هر نخ، بخشی از حاصل ضرب را بین بلوک‌های حافظه پنهان A و B محاسبه می‌کند. نتیجه در متغیر `tmp` جمع می‌شود.

پس از پردازش همه تکه‌ها، نخ ها دوباره همگام می‌شوند. نتیجه نهایی برای عنصر (ThreadRow، threadCol) در ماتریس C به صورت زیر محاسبه می‌شود:

C[threadRow * N + threadCol] = alpha*tmp + beta*C[threadRow * N + threadCol]

فراخوانی کرنل به صورت زیر است:

```
// Launch the kernel
        dim3 blockDim(32, 32, 1);
        dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
        size_t sharedMemSize = 2 * BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

        sgemm_shared_mem_block <<<grid, block, sharedMemSize >>> (M, N, K,
alpha, d_A, d_B, beta, d_C);
        cudaDeviceSynchronize();
```

مدت زمان لازم برای این عملیات با احتساب memoryAllocationها و انتقال‌های بین host و device برابر بود با:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time: 0.21420000000000000000
```

و بدون احتساب سایر مراحل و تنها زمان انجام محاسبات برروی GPU:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time: 0.08659999999999999600
```

گام سوم

شدت محاسباتی کرنل‌های ما تا کنون خیلی خوب نبوده است. می‌توانیم با سپردن محاسبه بیش از یک بخش به هر نخ نتیجه را بهبود دهیم. کرنل را طوری تغییر می‌دهیم که هر نخ نتایج موقت را برای مساحت ۸ در ۸ از بلوک‌ها در حافظه مشترک محاسبه کند. ابعاد بلوک حافظه مشترک را هم به B=8، BM=128 و BN=128 تغییر دادیم.

```
__global__ void __launch_bounds__((BM * BN) / (TM * TN), 1)
    sgemm2DBlocktiling(int M, int N, int K, float alpha, const float *A,
                       const float *B, float beta, float *C) {
  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);

  // 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
  extern __shared__ float As[BM * BK];
  extern __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
  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];
    }
  }
}
```

__launch_bounds__ حداکثر تعداد نخ‌ها در هر بلوک را برای این کرنل مشخص می‌کند.

پس عبارت (BM * BN) / (TM * TN) حداکثر تعداد نخ‌های مجاز در یک بلوک را تعیین می‌کند.

اندیس‌های نخ برای محاسبه اندیس‌های ردیف و ستون برای نخ فعلی استفاده می‌شوند و threadCol نشان‌دهنده ستون درون بلوک است و threadRow نشان‌دهنده ردیف داخل بلوک است.

آرایه‌های حافظه مشترک As و Bs را به ترتیب با اندازه‌های BM * BK و BK * BN اختصاص می‌دهیم. این آرایه‌ها برای کش کردن بخش‌هایی از ماتریس‌های A و B در حافظه مشترک استفاده می‌شوند.

پوینترهای A، B و C در شروع این بلوک تنظیم می‌شوند. پوینتر A به ابتدای سطر در ماتریس A مربوط به بلوک فعلی اشاره می‌کند. همینطور پوینتر B و C.

نخ‌ها عناصر A و B را در حافظه مشترک (As و Bs) بارگذاری می‌کنند.

این بار حلقه بیرونی بر روی تکه‌هایی به اندازه BK در امتداد بعد مشترک K تکرار می‌شود. در هر تکه، هر نخ بخشی از حاصل ضرب را بین بلوک‌های ذخیره شده A و B محاسبه می‌کند. نتیجه در آرایه threadResults جمع می‌شود.

نخ‌ها با استفاده از __syncthreads پس از بارگذاری داده‌ها و پس از تکمیل محاسبات همگام می‌شوند.

نتیجه نهایی برای عنصر (threadRow، threadCol) در ماتریس C به صورت زیر محاسبه می‌شود:

$$C[(threadRow*TM+resIdxM)*N+threadCol*TN+resIdxN]=\alpha\cdot threadResults[resIdxM*TN]$$
$$C[(threadRow*TM+resIdxM)*N+threadCol*TN+resIdxN]$$

فراخوانی کرنل به این صورت است:

```
// Launch the kernel
    dim3 blockDim(32, 32, 1);
    dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
    size_t sharedMemSize = 2 * BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

    sgemm2DBlocktiling <<<grid, block, sharedMemSize >>> (M, N, K, alpha, d_A,
d_B, beta, d_C);
    cudaDeviceSynchronize();
```

مدت زمان لازم برای این عملیات با احتساب memoryAllocationها و انتقال‌های بین host و device برابر بود با:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time: 0.12479999999999999000
```

و بدون احتساب سایر مراحل و تنها زمان انجام محاسبات برروی GPU:

```
H:\GPU8\HW6\x64\Release>HW6.exe
Averaged Elapsed Time: 0.00020000000000000001
```

Speedup هایی که در روش اول زمان‌گیری گرفتیم در هر مرحله به این قرار است:

T0/T1 = 5.104

T1/T2 = 1.52

T2/T3 = 1.72

Speedup هایی که در روش دوم زمان‌گیری گرفتیم در هر مرحله به این قرار است:

T0/T1 = 7.55

T1/T2 = 2.29

T2/T3 = 433

نهایتا شایان ذکر است که محاسبات ما تنها در ابعاد ۲۰۴۸ و با استفاده از کرنل‌های ۱، ۲، ۳ و ۵ از نمودار زیر انجام شد: