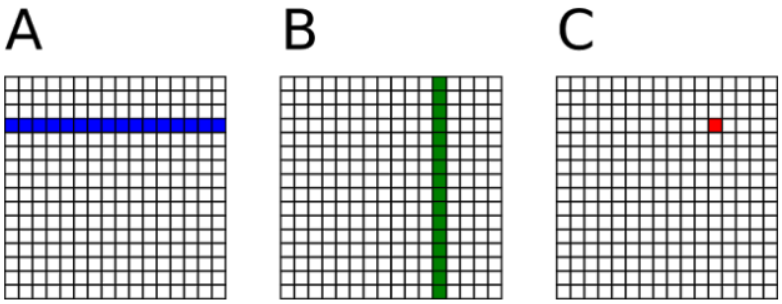
矩阵乘法的 CUDA 实现、优化及性能分析

矩阵乘法的CPU实现

```
// Matrix multiplication on the (CPU) host
void main(){
  define A, B, C
  for i = 0 to M-1 do
    for j = 0 to N-1 do
      /* compute element C(i,j) */
      for k = 0 to K-1 do
        C(i,j) <= C(i,j) + A(i,k) * B(k,j)
      end
    end
  end
}
```



CPU程序通过三层循环实现：

```
void  multiplicateMatrixOnHost(float *array_A, float *array_B, float *array_C, int M_p, int K_p, int N_p)
{
    for (int i = 0; i < M_p; i++)
    {
        for (int j = 0; j < N_p; j++)
        {
            float sum = 0;
            for (int k = 0; k < K_p; k++)
            {
                sum += array_A[i*K_p + k] * array_B[k*N_p + j];
            }
            array_C[i*N_p + j] = sum;
        }
    }

}
```

计算次数为 m × n × k 时间复杂度为 O(N^3) 获得C矩阵的计算方法都是相同的, 只不过使用的是矩阵A、B不同的元素来进行计算, 即不同数据的大量相同计算操作, 这种计算是特别适合使用GPU来计算, 因为GPU拥有大量简单重复的计算单元, 通过并行就能极大的提高计算效率.

下图是cpu计算矩阵乘法，在不同blocksize运行得到的秒数

m * n * k / blocksize	8 * 8	16 * 16	32 * 32
8 * 8 * 8	0.000004s	0.000005s	0.000004s
16 * 16 * 16	0.000027s	0.000018s	0.000032s
32 * 32 * 32	0.000186s	0.000210s	0.000167s
64 * 64 * 64	0.001036s	0.001034s	0.001257s
128 * 128 * 128	0.008488s	0.008442s	0.010756s
256 * 256 * 256	0.112544s	0.111550s	0.112556s
512 * 512 * 512	0.728302s	0.757964s	0.773689s
1024 * 1024 * 1024	5.974075s	6.044358s	5.665165s
2048 * 2048 * 2048	63.872070s	64.787750s	65.022842s
4096 * 4096 * 4096	686.902527s	688.189575s	754.218262s

矩阵乘法的 GPU 常规实现使用 Global Memory

在 GPU 中执行矩阵乘法运算操作：

在 Global Memory 中分别为矩阵 A、B、C 分配存储空间.

由于矩阵 C 中每个元素的计算均相互独立, NVIDIA GPU 采用的 SIMT (单指令多线程)的体系结构来实现并行计算的, 因此在并行度映射中, 我们让每个 thread 对应矩阵 C 中1个元素的计算.

执行配置 (execution configuration)中 gridSize 和 blockSize 均有 x(列向)、y(行向)两个维度. 其中,

```
gridsize.x * blocksize.x = n
gridsize.y * blcoksize.y = m
```

每个 thread 需要执行的 workflow 为：从矩阵 A 中读取一行向量 (长度为k), 从矩阵 B 中读取一列向量 (长度为k), 对这两个向量做点积运算 (单层 k 次循环的乘累加), 最后将结果写回矩阵 C.

CUDA的kernel函数实现如下：

```
__global__ void multiplicateMatrixOnDevice(float *array_A, float *array_B, float *array_C,
{
    int ix = threadIdx.x + blockDim.x*blockIdx.x;//row number
    int iy = threadIdx.y + blockDim.y*blockIdx.y;//col number

    if (ix < N_p && iy < M_p)
    {
        float sum = 0;
        for (int k = 0; k < K_p; k++)
        {
            sum += array_A[iy*K_p + k] * array_B[k*N_p + ix];
        }
        array_C[iy*N_p + ix] = sum;
    }
}
```

下面来分析一下该 kernel 函数中 A、B、C 三个矩阵对 global memory 的读取和写入情况：

读取 Global Memory：

对于矩阵 C 中每一个元素计算, 需要读取矩阵 A 中的一行元素;

对于矩阵 C 中同一行的 n 个元素, 需要重复读取矩阵 A 中同一行元素 n 次;

对于矩阵 C 中每一个元素计算, 需要读取矩阵 B 中的一列元素;

对于矩阵 C 中同一列的 m 个元素, 需要重复读取矩阵 B 中同一列元素 m 次;

写入 Global Memory：

矩阵 C 中的所有元素只需写入一次.
由此可见：

对 A 矩阵重复读取n次, 共计 m × k × n次 32bit Global Memory Load操作;
对 B 矩阵重复读取m次, 共计 k × n × m次 32bit Global Memory Load操作;
对 C 矩阵共计m × n次, 32bit Global Memory Store操作.

下图是gpu 在global memory计算矩阵乘法，在不同blocksize运行得到的秒数

m * n * k / blocksize	8 * 8	16 * 16	32 * 32
8 * 8 * 8	0.000031	0.000031	0.000024
16 * 16 * 16	0.000025	0.000040	0.000029
32 * 32 * 32	0.000027	0.000028	0.000028
64 * 64 * 64	0.000032	0.000043	0.000031
128 * 128 * 128	0.000067	0.000069	0.000064
256 * 256 * 256	0.000318	0.000323	0.000427
512 * 512 * 512	0.002301	0.002297	0.002302
1024 * 1024 * 1024	0.019319	0.019250	0.019231
2048 * 2048 * 2048	0.150697	0.150745	0.146186
4096 * 4096 * 4096	0.895327	0.871708	0.898963

结果分析：

- 随着矩阵规模增大，计算性能不断提升，到达峰值后又略有下降。在矩阵规模较小时，由于block数量不够多，无法填满所有SM单元，此时的性能瓶颈为Latency Bound（由于低Occupancy导致GPU计算资源的利用率低，延迟无法被很好的隐藏）；随着矩阵规模增加，block数量增加，每个SM中的active warps数量随之增大，此时Latency不再是性能瓶颈，转而受限于Memory Bound（过多的高延迟、低带宽的全局内存访问），在无法提升memory访问效率的情况下，性能无法进一步提升；
- 不同的blockSize对性能的影响不大（这里仅限于讨论88,1616,32*32三种情况）。究其原因，是因为我们选择的几种block维度设计（每行分别有8/16/32个thread），对1个warp内访问Global Memory时（Load或Store）transaction的数量没有变化

矩阵乘法的 GPU 常规实现使用 Shared Memory

虽然 warp 内对 Global Memory 的访问均已最大的实现了合并访问，但在 A、B 矩阵的读取操作中仍然有很多重复访问，例如，对于矩阵 A 的读取操作，通过合并访问（32 个 thread 访问 Global Memory 的同一地址，合并为一次访问），实际重复读取次数是(n/32); 对于矩阵 B 的读取操作，通过合并访问（8 个 thread 访问 32 Byte 数据可合并为一次），实际重复读取次数为(m/8)次。

在不改变这种数据读取方式的前提下又如何优化性能呢？在GPU中除了 Global Memory 还有 Shared Memory，这部分 Memory 是在芯片内部的，相较于 Global Memory 400~600 个时钟周期的访问延迟，Shared Memory 延时小 20-30 倍、带宽高 10 倍，具有低延时、高带宽的特性。因此性能优化的问题可以转变为如何利用 Shared Memory 代替 Global Memory 来实现数据的重复访问。

使用 Shared Memory 优化 Global Memory 访问的基本思想是充分利用数据的局部性。让一个 block 内的 thread 先从 Global Memory 中读取子矩阵块数据（大小为 BLOCK_SIZE*BLOCK_SIZE）并写入 Shared Memory 中; 在计算时，从 Shared Memory 中（重复）读取数据做乘累加，从而避免每次都到 Global 中取数据带来的高延迟影响。接下来让子矩阵块分别在矩阵 A 的行向以及矩阵 B 的列向上滑动，直到计算完所有k个元素的乘累加。

m * n * k / blocksize	8 * 8	16 * 16	32 * 32
8 * 8 * 8	0.000013s	0.000015s	0.000015s
16 * 16 * 16	0.000014s	0.000013s	0.000014s
32 * 32 * 32	0.000015s	0.000014s	0.000014s
64 * 64 * 64	0.000019s	0.000017s	0.000017s
128 * 128 * 128	0.000048s	0.000038s	0.000038s
256 * 256 * 256	0.000273s	0.000204s	0.000191s
512 * 512 * 512	0.001953s	0.001443s	0.001423s
1024 * 1024 * 1024	0.017410s	0.017059s	0.011220s
2048 * 2048 * 2048	0.094129s	0.080054s	0.081923s
4096 * 4096 * 4096	0.677808s	0.494800s	0.491805s

```
// Compute C = A * B
__global__ void matrixMultiplyShared(float *A, float *B, float *C,
    int numARows, int numAColumns, int numBRows, int numBColumns, int numCRows, int numCColumns)
{
    //@@ Insert code to implement matrix multiplication here
    //@@ You have to use shared memory for this MP

    __shared__ float sharedM[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float sharedN[BLOCK_SIZE][BLOCK_SIZE];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;


    int row = by * BLOCK_SIZE + ty;
    int col = bx * BLOCK_SIZE + tx;
```

矩阵乘法的 GPU 常规实现使用Titled shared memeory，调用Cublas

前面的算法设计中，每个线程只计算了矩阵 C 中的一个元素，每个线程每个内层循环需要从子矩阵 A 和子矩阵 B 中各读取一个 4Byte 的元素（共取 8Byte 数据执行2次浮点运算），实际上我们可以让每个

线程读取一组 Shared Memory 数据后（放入寄存器中），计算更多的元素，从而减少 Shared Memory 的访问。

```
float a = 1, b = 0;
cublasSgemm(
  handle,
  CUBLAS_OP_T,   //矩阵A的属性参数，转置，按行优先
  CUBLAS_OP_T,   //矩阵B的属性参数，转置，按行优先
  M,          //矩阵A、C的行数
  N,          //矩阵B、C的列数
  K,          //A的列数，B的行数，此处也可为B_ROW,一样的
  &a,             //alpha的值
  d_A,            //左矩阵，为A
  K,          //A的leading dimension，此时选择转置，按行优先，则leading dimension为A的列数
  d_B,            //右矩阵，为B
  N,          //B的leading dimension，此时选择转置，按行优先，则leading dimension为B的列数
  &b,             //beta的值
  d_C,            //结果矩阵C
  M           //C的leading dimension，C矩阵一定按列优先，则leading dimension为C的行数
);
cudaMemcpy(deviceRef, d_C, Cxy * sizeof(float), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
cudaEventRecord(gpustop, 0);
cudaEventSynchronize(gpustop);

cudaEventElapsedTime(&elapsedTime, gpustart, gpustop);
cudaEventDestroy(gpustart);
cudaEventDestroy(gpustop);

printf("Matrix_deviceRef: (%d×%d)  <<<(%d,%d),(%d,%d)>>>  GPU运行时间为：%fs\n",
       M, N, grid.x, grid.y, block.x, block.y, elapsedTime / 1000);
```

上面的 kernel 函数中，注意观察内层循环：我们让 1 个 thread 分别从子矩阵 A 中读取 2 个数据，从子矩阵 B 中读取 1 个数据（注意2次取数据是同一地址！），然后同时计算2个元素 val[0] 和 val[1]。此时，通过读取 4B*3 个数据，实现了2次乘加共4次浮点计算。减少了 shared memory 中子矩阵B一半的数据访问。

m * n * k / blocksize	8 * 8	16 * 16	32 * 32
8 * 8 * 8	0.000013s	0.000015s	0.000015s
16 * 16 * 16	0.000014s	0.000013s	0.000014s
32 * 32 * 32	0.000015s	0.000014s	0.000014s
64 * 64 * 64	0.000019s	0.000017s	0.000017s
128 * 128 * 128	0.000048s	0.000038s	0.000038s
256 * 256 * 256	0.000273s	0.000204s	0.000191s
512 * 512 * 512	0.001953s	0.001443s	0.001423s
1024 * 1024 * 1024	0.017410s	0.017059s	0.011220s
2048 * 2048 * 2048	0.094129s	0.080054s	0.081923s
4096 * 4096 * 4096	0.677808s	0.494800s	0.491805s

Shared Memory Load：在每 1 次内层循环中，1 个 warp 内的 32 个 thread 需要从 shTileA 读取同 2 个元素，需要 2 次 Shared Memory Load Transactions，再从 shTileB 中读取连续的 32 个元素（假设没有 Bank Conflict，需要1次 Shared Memory Load Transactions）（注意val[0]和val[1]的计算中，shTileB 的地址是一样的），即总共需要 3 次 Shared Memory Load Transactions。

对GPU三个版本，从图中可以看出：

1. 在句子规模为 4K * 4K * 4K 的情况下， cublas 性能更高
2. 随着矩阵规模的增加， 计算性能也逐渐增加
3. 通过利用shared memory和寄存器能有效的降低IO带宽对性能的影响，从而高效的利用对GPU的硬件计算资源。