cuda编程实验

> 计算机体系架构 cuda编程实验
>
> 2022.5.30
>
> 无97 杨希杰 2019011170

准备命令

```
$ ssh tsinghuaee281@10.20.101.27
$ ssh-copy-id -i ~/.ssh/id_rsa.pub tsinghuaee281@10.20.101.27
$ scp -r /Users/yangxijie/yxj/SEMESTER6/C\ CS/实验/Cuda作业/MatrixMultiplication_hw tsinghuaee281@10.20.101.27:~/cudahw/0

$ module add cuda91/toolkit/9.1.85
```

1 - Tile size 1×1 per thread

核心代码

```
for(int k = 0; k < A.width; k++) {
    sum += A.elements[idx_y * A.width + k] * B.elements[k * B.width + idx_x];
}
```

运行结果

```
$ make cuda
$ for i in 1 2 3 4 5; do ./matrix_multiplication; done
CUDA Elapsed time: 3.425792 ms
Success!
CUDA Elapsed time: 3.452032 ms
Success!
CUDA Elapsed time: 3.533216 ms
Success!
CUDA Elapsed time: 3.417024 ms
Success!
CUDA Elapsed time: 3.453568 ms
Success!
```

2 - Increasing tile size per thread

核心代码

```
for (int k = 0; k < A.width; k++) {
    for (int i = 0; i < TILE_SIZE; i++) {
        a_vec[i] = A.elements[(tile_row + i) * A.width + k];
        b_vec[i] = B.elements[k * B.width + (tile_col + i)];
    }
    for (int r = 0; r < TILE_SIZE; r++) {
        for (int c = 0; c < TILE_SIZE; c++) {
            Csum[r][c] += a_vec[r] * b_vec[c];
        }
    }
}
```

可以看到这里将矩阵数据按一个小向量打包读取，提高了算存比，增大了数据复用

运行结果

```
# TILE_SIZE = 1
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
(ms) (3.440544 + 3.434144 + 3.468192 + 3.423360 + 3.433440) / 5 = 3.440
# TILE_SIZE = 2
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
(ms) (2.772032 + 2.765888 + 2.737216 + 2.872800 + 2.752608) / 5 = 2.780
# TILE_SIZE = 4
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
(ms) (2.561184 + 2.663968 + 2.567392 + 2.567200 + 2.589536) / 5 = 2.590
# TILE_SIZE = 8
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
(ms) (3.533632 + 3.523648 + 3.593344 + 3.567008 + 3.451136) / 5 = 3.534
# TILE_SIZE = 16
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
(ms) (8.452352 + 8.379648 + 8.386816 + 8.361056 + 8.377408) / 5 = 8.391
# TILE_SIZE = 32
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
```

```
(ms) (27.050304 + 27.187263 + 27.233791 + 27.108671 + 27.085184) / 5 = 27.13
# TILE_SIZE = 64
$ make clean && make cuda_method2 && for i in 1 2 3 4 5; do
./matrix_multiplication_method2; done
(ms) (98.427170 + 98.351425 + 98.505630 + 99.080605 + 99.031837) / 5 = 98.68
```

可以看到：随着 `TILE_SIZE` 的增大，刚开始执行速度有很大提升，这是由于算存比提高使得读取数据不那么频繁；但随着 `TILE_SIZE` 进一步提升，由于GPU并行度降低，因此执行速度降低。这启发我们在进行并行计算的时候，需要在数据与并行之间找到一个最优点。

3 - Optimization using shared memory

核心代码

```
for (int m = 0; m < (A.width / BLOCK_SIZE); m++) {
    // Use GetSubMatrix function to get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);

    // Use GetSubMatrix function to get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);

    // Shared memory used to store Asub and Bsub respectively
    __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix.
    // You can use GetElement function.
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    // Each thread computes ONE element of Csub
    // by accumulating results into Cvalue
    for (int ii = 0; ii < BLOCK_SIZE; ii++) {
        Cvalue += As[row][ii] * Bs[ii][col];
    }

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}
```

这里需要注意的是：这样的一个函数块只计算一个tile的值，在加载数据的过程中，也只是加载一个位置上的数据。

运行结果

```
# BLOCK_SIZE = 1
$ make clean && make cuda_opt && for i in 1 2 3 4 5; do
./matrix_multiplication_opt; done
(ms) (51.712097 + 51.725086 + 51.678879 + 51.765663 + 51.833473) / 5 = 51.74
# BLOCK_SIZE = 2
$ make clean && make cuda_opt && for i in 1 2 3 4 5; do
./matrix_multiplication_opt; done
(ms) (9.067776 + 9.069152 + 9.130688 + 9.096960 + 9.095712) / 5 = 9.09
# BLOCK_SIZE = 4
$ make clean && make cuda_opt && for i in 1 2 3 4 5; do
./matrix_multiplication_opt; done
(ms) (3.275680 + 3.224512 + 3.325344 + 3.240224 + 3.241024) / 5 = 3.26
# BLOCK_SIZE = 8
$ make clean && make cuda_opt && for i in 1 2 3 4 5; do
./matrix_multiplication_opt; done
(ms) (2.395712 + 2.349440 + 2.401856 + 2.404064 + 2.350624) / 5 = 2.38
# BLOCK_SIZE = 16
$ make clean && make cuda_opt && for i in 1 2 3 4 5; do
./matrix_multiplication_opt; done
(ms) (2.411744 + 2.398976 + 2.359040 + 2.380992 + 2.367104) / 5 = 2.38
# BLOCK_SIZE = 32
$ make clean && make cuda_opt && for i in 1 2 3 4 5; do
./matrix_multiplication_opt; done
(ms) (2.497248 + 2.368448 + 2.373824 + 2.386144 + 2.430048) / 5 = 2.41
```

可以看到，通过共享计算单元数据的方式，极大的提高了数据加载的效率，这使得运算的速度几乎只取决于并行的线程数，在当前的方案中，由于 `TILE_SIZE` 都为 `1`，因此当 `BLOCK_SIZE` 大于一定程度时，运算所花费的时间基本稳定。