

**Задание:**

Реализовать транспонирование матрицы размерностью N\*K, где = 8\*2^12, число нитей взято threadsPerBlock = 128, использования разделяемой памяти, с разделяемой памятью без разрешения конфликта банков и с разрешением конфликта банков. Сравнить время выполнения соответствующих ядер на GPU. Для всех трёх случаев определить эффективность использования разделяемой памяти с помощью метрик nvprof или ncu.

**Цель:** приобретение навыков использования разделяемой памяти.

**Выполнение работы:**

Для выполнения работы была написана программа, реализующая транспонирование матрицы тремя методами:

* без использования shared памяти
* с использованием shared памяти и с возникновение конфликта банков
* с использование shared памяти и решением конфликта памяти

| #include <iostream> #include <cstdlib> #include "cuda\_runtime.h" #include "device\_launch\_parameters.h"  using namespace std;  #define CUDA\_NUM 32  \_\_global\_\_ void gBase\_Transposition(float \*matrix, float \*result, const int N, const int K) {  unsigned int k = threadIdx.x + blockIdx.x \* blockDim.x;  unsigned int n = threadIdx.y + blockIdx.y \* blockDim.y;  result[n + k \* N] = matrix[k + n \* K]; } \_\_global\_\_ void gShared\_Transposition\_Wrong(float \*matrix, float \*result, const int N, const int K) {  \_\_shared\_\_ float shared[CUDA\_NUM][CUDA\_NUM];  unsigned int k = threadIdx.x + blockIdx.x \* blockDim.x;  unsigned int n = threadIdx.y + blockIdx.y \* blockDim.y;   shared[threadIdx.y][threadIdx.x] = matrix[K + n \* N];  \_\_syncthreads();   k = threadIdx.x + blockIdx.y \* blockDim.x;  n = threadIdx.y + blockIdx.x \* blockDim.y;  result[k + n \* N] = shared[threadIdx.x][threadIdx.y]; } \_\_global\_\_ void gShared\_Transposition(float \*matrix, float \*result, const int N, const int K) {  \_\_shared\_\_ float shared[CUDA\_NUM][CUDA\_NUM + 1];  unsigned int k = threadIdx.x + blockIdx.x \* blockDim.x;  unsigned int n = threadIdx.y + blockIdx.y \* blockDim.y;   shared[threadIdx.y][threadIdx.x] = matrix[K + n \* N];  \_\_syncthreads();   k = threadIdx.x + blockIdx.y \* blockDim.x;  n = threadIdx.y + blockIdx.x \* blockDim.y;  result[k + n \* N] = shared[threadIdx.x][threadIdx.y]; } void MatrixShow(const int N, const int K, const float \*Matrix) {  cout << endl;  for (long long i = 0; i < K; ++i) {  for (long long j = 0; j < N; ++j) {  cout << Matrix[j + i \* N] << " ";  }  cout << endl;  }  cout << endl; }  int main() {  const int num = 1 << 12;  int N = 8 \* num, K = 8 \* num, threadsPerBlock = 128;  float \*GPU\_pre\_matrix, \*local\_pre\_matrix, \*GPU\_after\_matrix, \*local\_after\_matrix, elapsedTime;  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);   /\* простое транспонирование \*/   cudaMalloc((void \*\*) &GPU\_pre\_matrix, N \* K \* sizeof(float));  cudaMalloc((void \*\*) &GPU\_after\_matrix, N \* K \* sizeof(float));   local\_pre\_matrix = (float \*) calloc(N \* K, sizeof(float));  local\_after\_matrix = (float \*) calloc(N \* K, sizeof(float));   for (int i = 0; i < N; ++i) {  for (int j = 0; j < K; ++j) {  local\_pre\_matrix[j + i \* K] = j + i \* K + 1;  }  }   cudaMemcpy(GPU\_pre\_matrix, local\_pre\_matrix, K \* N \* sizeof(float), cudaMemcpyHostToDevice);   cudaEventRecord(start, nullptr);  gBase\_Transposition <<< dim3(K / threadsPerBlock,N / threadsPerBlock),  dim3(threadsPerBlock, threadsPerBlock) >>>  (GPU\_pre\_matrix, GPU\_after\_matrix, N, K);  cudaDeviceSynchronize();  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);   cudaMemcpy(local\_after\_matrix, GPU\_after\_matrix, K \* N \* sizeof(float), cudaMemcpyDeviceToHost);  cudaEventElapsedTime(&elapsedTime, start, stop);   cout<<"1st method Matrix: "<<endl;   cout << "gBase\_Transposition:\n\t"  << elapsedTime  << endl;   cudaFree(GPU\_after\_matrix);  free(local\_after\_matrix);   /\* транспонирование без решения проблемы конфликта банков \*/   cudaMalloc((void \*\*) &GPU\_after\_matrix, N \* K \* sizeof(float));  local\_after\_matrix = (float \*) calloc(N \* K, sizeof(float));   cudaEventRecord(start, nullptr);  gShared\_Transposition\_Wrong <<< dim3(K / threadsPerBlock, N / threadsPerBlock),  dim3(threadsPerBlock,threadsPerBlock) >>>  (GPU\_pre\_matrix, GPU\_after\_matrix, N, K);   cudaDeviceSynchronize();  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);   cudaMemcpy(local\_after\_matrix, GPU\_after\_matrix, K \* N \* sizeof(float), cudaMemcpyDeviceToHost);  cudaEventElapsedTime(&elapsedTime, start, stop);   cout<<"2st method Matrix: "<<endl;  cout << "gShared\_Transposition\_Wrong:\n\t"  << elapsedTime  << endl;   cudaFree(GPU\_after\_matrix);  free(local\_after\_matrix);   /\* транспонирование с решением проблемы конфликта банков \*/   cudaMalloc((void \*\*) &GPU\_after\_matrix, N \* K \* sizeof(float));  local\_after\_matrix = (float \*) calloc(N \* K, sizeof(float));   cudaEventRecord(start, nullptr);  gShared\_Transposition <<< dim3(K / threadsPerBlock, N / threadsPerBlock),  dim3(threadsPerBlock,threadsPerBlock) >>>  (GPU\_pre\_matrix, GPU\_after\_matrix, N, K);   cudaDeviceSynchronize();  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);   cudaMemcpy(local\_after\_matrix, GPU\_after\_matrix, K \* N \* sizeof(float), cudaMemcpyDeviceToHost);  cudaEventElapsedTime(&elapsedTime, start, stop);   cout<<"3st method Matrix: "<<endl;   cout << "gShared\_Transposition:\n\t"  << elapsedTime  << endl;   cudaFree(GPU\_pre\_matrix);  cudaFree(GPU\_after\_matrix);  free(local\_pre\_matrix);  free(local\_after\_matrix);   return 0; } |
| --- |

Листинг 1 – main.cu

Команда компиляции и результат работы программы:

| D:\Projects\CUDA\_CMake\cmake-build-debug\LR05\_GPU.exe 1st method Matrix: gBase\_Transposition:  35.7433 2st method Matrix: gShared\_Transposition\_Wrong:  0.154592 3st method Matrix: gShared\_Transposition:  0.43648  Process finished with exit code 0 |
| --- |

Результат nvprof:

| dany@DESKTOP-UMC1Q46:/mnt/d/Projects/CUDA\_CMake/LR05/src$ nvcc LR05\_1G.cu dany@DESKTOP-UMC1Q46:/mnt/d/Projects/CUDA\_CMake/LR05/src$ nvprof ./a.out ==3237== NVPROF is profiling process 3237, command: ./a.out ==3237== Warning: Unified Memory Profiling is not supported on the current configuration because a pair of devices without peer-to-peer support is detected on this multi-GPU setup. When peer mappings are not available, system falls back to using zero-copy memory. It can cause kernels, which access unified memory, to run slower. More details can be found at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-managed-memory 1st method Matrix: gBase\_Transposition:  0.469216 2st method Matrix: gShared\_Transposition\_Wrong:  0.166496 3st method Matrix: gShared\_Transposition:  0.172192 ==3237== Profiling application: ./a.out ==3237== Profiling result:  Type Time(%) Time Calls Avg Min Max Name  GPU activities: 94.98% 40.2303s 3 13.4101s 1.98314s 33.6600s [CUDA memcpy DtoH]  5.02% 2.12479s 1 2.12479s 2.12479s 2.12479s [CUDA memcpy HtoD]  API calls: 89.49% 48.6340s 4 12.1585s 1.98407s 33.6653s cudaMemcpy  4.19% 2.27843s 3 759.48ms 230.07us 1.14023s cudaDeviceSynchronize  3.21% 1.74246s 4 435.62ms 139.11ms 644.37ms cudaFree  2.06% 1.12042s 2 560.21ms 1.1630us 1.12042s cudaEventCreate  1.03% 558.67ms 4 139.67ms 47.523ms 228.91ms cudaMalloc  0.01% 6.6275ms 1 6.6275ms 6.6275ms 6.6275ms cuDeviceGetPCIBusId  0.00% 1.4618ms 3 487.28us 5.2800us 1.4513ms cudaEventElapsedTime  0.00% 452.32us 6 75.387us 36.497us 136.04us cudaEventRecord  0.00% 408.92us 3 136.31us 1.3630us 355.82us cudaLaunchKernel  0.00% 189.85us 3 63.283us 43.901us 98.552us cudaEventSynchronize  0.00% 19.406us 101 192ns 130ns 1.1220us cuDeviceGetAttribute  0.00% 2.2040us 3 734ns 320ns 1.1630us cuDeviceGetCount  0.00% 1.3320us 2 666ns 240ns 1.0920us cuDeviceGet  0.00% 1.0220us 1 1.0220us 1.0220us 1.0220us cuDeviceGetName  0.00% 431ns 1 431ns 431ns 431ns cuDeviceTotalMem  0.00% 241ns 1 241ns 241ns 241ns cuDeviceGetUuid |
| --- |

По результатам работы программы можно сделать вывод, что использование shared памяти положительно влияет на скорость выполнения задачи, но в случае не решенной проблемы конфликта банков, а именно когда происходит запись в одну и туже ячейку идет потеря производительности. С использование shared памяти +1 эта проблема исчезает.

**Вывод:**   
В ходе выполнения работы, была исследована и применена работа с глобальной памятью графического процессора (GPU) с использованием технологии CUDA. В ходе работы мы ознакомились с работой с shared памятью и разрешением конфликта банков памяти.