Кафедра прикладной математики и кибернетики

Выполнили:

Студенты 3 курса группы ИП-111  
Корнилов А.А.,  
Попов М.И.,

Толкач А.А.

Проверил:

Профессор кафедры ПМиК  
Малков Е.А.

ЛАБОРАТОРНАЯ РАБОТА №8

По дисциплине: «Программирование графических процессоров»

Сибирский государственный университет  
телекоммуникаций и информатики

Новосибирск, 2024

Министерство цифрового развития, связи  
и массовых коммуникаций Российской Федерации

**Задание:**

* Реализовать вычисление на GPU произведения матриц, используя CUDA API (“сырой код”) и, отдельно, используя библиотеку cuBLAS. Сравнить время выполнения программ при различной размерности матриц.

**Цель:** освоить использование библиотеки cuBLAS.

**Оборудование**: Видеокарта GTX 1050TI (Pascal)

**Выполнение работы:**

Для выполнения работы была написана программа с использование функции cublasSgemm для перемножения матриц, в отличии от других программ для CUDA для cuBLAS нужно объявлять её дескриптор в нашем случае (cublasHandle\_t handle), далее слеует обычное выделение памяти и вызов функции cublasSgemm в которой мы указываем CUBLAS\_OP\_N для указывает что не нужно транспонировать для умножения, размеры матриц NNN, скаляр для первой матрицы alpha равный 1.0f – то есть \* на 1 входную матрицу, указываем входные матрицы d\_A и d\_B и их размеры NN, beta выходной скаляр который прибавляется к выходной матрице, выходная матрица d\_C и её размер N :

|  |
| --- |
| #include <iostream> #include <iomanip> #include <cuda\_runtime.h> #include <cublas\_v2.h>  void initMatrix(float \*matrix, int rows, int cols) {  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < cols; ++j) {  matrix[i \* cols + j] = i + j;  }  } } void printMatrix(float \*matrix, int rows, int cols) {  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < cols; ++j) {  std::cout << matrix[i \* cols + j] << "\t";  }  std::cout << std::endl;  } }  int main() {  const int num = 1 << 2;  int N = 3 \* num;  float elapsedTime = 0;  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);   float \*h\_A = new float[N \* N];  float \*h\_B = new float[N \* N];  float \*h\_C = new float[N \* N];   initMatrix(h\_A, N, N);  initMatrix(h\_B, N, N);   cublasHandle\_t handle;  **cublasCreate**(&handle);   float \*d\_A, \*d\_B, \*d\_C;  cudaMalloc(&d\_A, N \* N \* sizeof(float));  cudaMalloc(&d\_B, N \* N \* sizeof(float));  cudaMalloc(&d\_C, N \* N \* sizeof(float));   cudaMemcpy(d\_A, h\_A, N \* N \* sizeof(float), cudaMemcpyHostToDevice);  cudaMemcpy(d\_B, h\_B, N \* N \* sizeof(float), cudaMemcpyHostToDevice);   float alpha = 1.0f, beta = 0.0f;   cudaEventRecord(start, 0);  **cublasSgemm**(handle, CUBLAS\_OP\_N, CUBLAS\_OP\_N, N, N, N, &alpha, d\_A, N, d\_B, N, &beta, d\_C, N);  cudaDeviceSynchronize();  cudaEventRecord(stop, 0);   cudaEventSynchronize(stop);  cudaEventElapsedTime(&elapsedTime, start, stop);  std::cout << "Time using cuBLAS code: " << std::setprecision(15) << elapsedTime << std::endl;   cudaMemcpy(h\_C, d\_C, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);   std::cout << "Matrix A:" << std::endl;  printMatrix(h\_A, N, N);  std::cout << std::endl;   std::cout << "Matrix B:" << std::endl;  printMatrix(h\_B, N, N);  std::cout << std::endl;   std::cout << "End matrix C:" << std::endl;  printMatrix(h\_C, N, N);   cudaFree(d\_A);  cudaFree(d\_B);  cudaFree(d\_C);  **cublasDestroy**(handle);  delete[] h\_A;  delete[] h\_B;  delete[] h\_C;  return 0; } |

Листинг 1 – программа LR08\_1.cu

Команда компиляции и результат работы программы:

|  |
| --- |
| PS D:\Projects\CUDA\_CMake\LR08\src> nvcc .\LR08\_1G.cu -lcublas  LR08\_1G.cu  tmpxft\_00042630\_00000000-10\_LR08\_1G.cudafe1.cpp  Создается библиотека a.lib и объект a.exp  PS D:\Projects\CUDA\_CMake\LR08\src> .\a.exe  Time using cuBLAS code: 0.647104024887085  Matrix A:  0 1 2 3 4 5 6 7 8  1 2 3 4 5 6 7 8 9  2 3 4 5 6 7 8 9 10  3 4 5 6 7 8 9 10 11  4 5 6 7 8 9 10 11 12  5 6 7 8 9 10 11 12 13  6 7 8 9 10 11 12 13 14  7 8 9 10 11 12 13 14 15  8 9 10 11 12 13 14 15 16  Matrix B:  0 1 2 3 4 5 6 7 8  1 2 3 4 5 6 7 8 9  2 3 4 5 6 7 8 9 10  3 4 5 6 7 8 9 10 11  4 5 6 7 8 9 10 11 12  5 6 7 8 9 10 11 12 13  6 7 8 9 10 11 12 13 14  7 8 9 10 11 12 13 14 15  8 9 10 11 12 13 14 15 16  End matrix C:  204 240 276 312 348 384 420 456 492  240 285 330 375 420 465 510 555 600  276 330 384 438 492 546 600 654 708  312 375 438 501 564 627 690 753 816  348 420 492 564 636 708 780 852 924  384 465 546 627 708 789 870 951 1032  420 510 600 690 780 870 960 1050 1140  456 555 654 753 852 951 1050 1149 1248  492 600 708 816 924 1032 1140 1248 1356 |

Также была написана программа используя «сырые» компоненты CUDA для перемножения матриц.

|  |
| --- |
| #include <iostream> #include <cuda\_runtime.h> #include <iomanip>  **\_\_global\_\_** void matrixMultiply(float \*a, float \*b, float \*c, int n) {  int row = blockIdx.y \* blockDim.y + threadIdx.y;  int col = blockIdx.x \* blockDim.x + threadIdx.x;   if (row < n && col < n) {  int sum = 0;  for (int i = 0; i < n; ++i) {  sum += a[row \* n + i] \* b[i \* n + col];  }  c[row \* n + col] = sum;  } } void initMatrix(float \*matrix, int rows, int cols) {  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < cols; ++j) {  matrix[i \* cols + j] = i + j;  }  } } void printMatrix(float \*matrix, int rows, int cols) {  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < cols; ++j) {  std::cout << matrix[i \* cols + j] << "\t";  }  std::cout << std::endl;  } }  int main() {  const int num = 1 << 12;  int N = 3 \* num;  float elapsedTime = 0;  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);   float \*h\_A = new float[N \* N];  float \*h\_B = new float[N \* N];  float \*h\_C = new float[N \* N];   initMatrix(h\_A, N, N);  initMatrix(h\_B, N, N);   float \*d\_A, \*d\_B, \*d\_C;  cudaMalloc(&d\_A, N \* N \* sizeof(float));  cudaMalloc(&d\_B, N \* N \* sizeof(float));  cudaMalloc(&d\_C, N \* N \* sizeof(float));   cudaMemcpy(d\_A, h\_A, N \* N \* sizeof(float), cudaMemcpyHostToDevice);  cudaMemcpy(d\_B, h\_B, N \* N \* sizeof(float), cudaMemcpyHostToDevice);   dim3 blockSize(1024, 1024);  dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) / blockSize.y);   cudaEventRecord(start, 0);  matrixMultiply<<<gridSize, blockSize>>>(d\_A, d\_B, d\_C, N);  cudaDeviceSynchronize();  cudaEventRecord(stop, 0);   cudaEventSynchronize(stop);  cudaEventElapsedTime(&elapsedTime, start, stop);  std::cout << "Time using CUDA code: " << std::setprecision(15) << elapsedTime << std::endl;   cudaMemcpy(h\_C, d\_C, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);  /\* std::cout << "Matrix A:" << std::endl;  printMatrix(h\_A, N, N);  std::cout << std::endl;   std::cout << "Matrix B:" << std::endl;  printMatrix(h\_B, N, N);  std::cout << std::endl;   std::cout << "End matrix C:" << std::endl;  printMatrix(h\_C, N, N);\*/   cudaFree(d\_A);  cudaFree(d\_B);  cudaFree(d\_C);  delete[] h\_A;  delete[] h\_B;  delete[] h\_C;  return 0; } |

Результат работы программы:

|  |
| --- |
| PS D:\Projects\CUDA\_CMake\LR08\src> nvcc .\LR08\_2G.cu  LR08\_2G.cu  tmpxft\_00004d9c\_00000000-10\_LR08\_2G.cudafe1.cpp  Создается библиотека a.lib и объект a.exp  PS D:\Projects\CUDA\_CMake\LR08\src> .\a.exe  Time using CUDA code: 0.526880025863647 |

К сожалению nvprof не получилось применить к программе.

|  |  |  |
| --- | --- | --- |
|  | CUDA | cuBLAS |
| 1 << 2 | 0.411648005247116 | 0.721920013427734 |
| 1 << 4 | 0.485983995199203 | 0.650367975234985 |
| 1 << 6 | 0.843871994018555 | 1.64681601524353 |
| 1 << 8 | 1.406495988368988 | 5.50540781021118 |
| 1 << 10 | 2.516704022884369 | 38.0118408203125 |
| 1 << 12 | 8.503935992717743 | 1456.44482421875 |

В результате программы мы видим что с повышение размера матрицы, время работы с cuBLAS повышается

Вывод: в ходе выполнения лабораторной работы, была исследована и применена работа с библиотекой cuBLAS, стало понятно что использование сырого CUDA кода быстрее при выполнении, в случаи cuBLAS время выполнения до 2^8 не сильно выше чем обычный CUDA код и для написание универсального кода её можно использовать если в нем не используется матрицы размеров 2^8.