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 №9

По дисциплине: «Программирование графических процессоров»

Сибирский государственный университет  
телекоммуникаций и информатики

Новосибирск, 2024

Министерство цифрового развития, связи  
и массовых коммуникаций Российской Федерации

**Задание:**

* разработать и программно реализовать алгоритм для сравнения производительности копирования устройство->хост (и наоборот) данных, размещенных в памяти выделенной на хосте обычным образом и с использованием закрепленных страниц ;
* подобрать оптимальный размер порции данных для реализации сложения векторов с использованием распараллеливания копирования и выполнения на основе потоков CUDA;
* то же для реализации скалярного умножения.

**Цель:** изучить преимущества использования потоков CUDA.

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

**Выполнение работы:**

Для выполнения работы была написана программа для сравнения скорости копирования вектора используя стандартное копирование и используя закрепленную память с устройства на хост и обратно.

|  |
| --- |
| #include <iostream> #include <cstdlib> #include <cuda\_runtime.h>  using namespace std;  int main() {  int num = 1 << 12;  int size = 32 \* num;  float \*device, \*hostPinned, \*host, time = 0;  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);   host = (float \*) malloc(size \* sizeof(float));  cudaMallocHost((void \*\*) &hostPinned, size \* sizeof(float));  cudaMalloc((void \*\*) &device, size \* sizeof(float));  cudaMemset(device, 1024, size \* sizeof(float));  cudaMemcpy(device, host, size \* sizeof(float), cudaMemcpyHostToDevice);   cudaEventRecord(start, nullptr);  cudaMemcpy(host, device, size \* sizeof(float), cudaMemcpyDeviceToHost);  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);  cudaEventElapsedTime(&time, start, stop);  cout << "Стандартное копирование с device на host: "<< time << endl;   cudaStream\_t stream;  cudaStreamCreate(&stream);  cudaEventRecord(start, nullptr);  cudaMemcpyAsync(hostPinned, device, size \* sizeof(float), cudaMemcpyDeviceToHost, stream);  cudaStreamSynchronize(stream);  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);  time = 0;  cudaEventElapsedTime(&time, start, stop);  cout << "Закрепленная память (pinned memory) с device на хост: "<< time << endl;   cudaEventRecord(start, nullptr);  cudaMemcpy(device, host, size \* sizeof(float), cudaMemcpyHostToDevice);  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);  time = 0;  cudaEventElapsedTime(&time, start, stop);  cout << "Стандартное копирование с host на device: " << time << endl;   cudaEventRecord(start, nullptr);  cudaMemcpyAsync(device, hostPinned, size \* sizeof(float), cudaMemcpyHostToDevice, stream);  cudaStreamSynchronize(stream);  cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);  time = 0;  cudaEventElapsedTime(&time, start, stop);  cout << "Закрепленная память (pinned memory) с хост на device: " << time << endl;   cudaFree(device);  cudaFreeHost(host);  free(host);  return 0; } |

Листинг 1 – программа LR09\_G1.cu

Команда компиляции и результат работы программы:

|  |
| --- |
| Стандартное копирование с device на host: 0.347648  Закрепленная память (pinned memory) с device на хост: 0.093056  Стандартное копирование с host на device: 0.139168  Закрепленная память (pinned memory) с хост на device: 0.088032  Process finished with exit code 0 |

В результате использование копирования используя стандартное копирование работает медлен при копировании с устройства на хост чем при использовании закреплённой памяти, и также наоборот копирование с хост на устройство выигрывает стандартная память

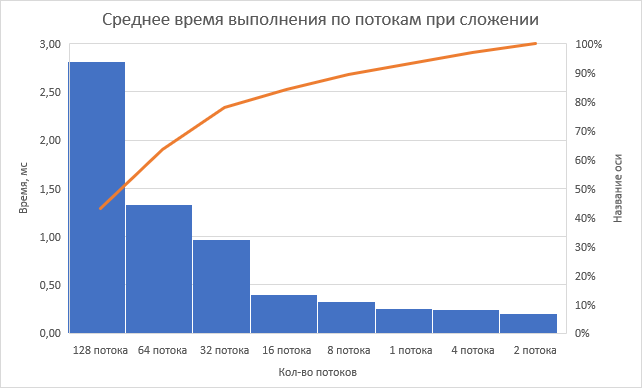
Для второго задания написана программа для сложения векторов используя закреплённую память и cudastream. Используеться деление на потоки от размера portion\_size, размер взят 32\*2^12

|  |
| --- |
| #include <iostream> #include <cuda.h> #include <cuda\_runtime.h>  using namespace std;  void show\_mass(float \*a, int num){  for (int i = 0; i < num; i++) {  printf("%f ",a[i]);  if (i%10 == 0) printf("\n");  }  printf("\n"); } **\_\_global\_\_** void addVectors(float \*a, float \*b, float \*c, int n) {  unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;  if (i < n){  c[i] = a[i] + b[i];  } }  int main() {  int stream\_num = 1;  int num = 1 << 12;  int size = 32 \* num;  int portion\_size = size / stream\_num;   float \*h\_a, \*h\_b, \*h\_c;  float \*d\_a, \*d\_b, \*d\_c;  float time = 0;  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);   auto \*streams = (cudaStream\_t\*)calloc(stream\_num, sizeof(cudaStream\_t));  for (int i = 0; i < stream\_num; i++) cudaStreamCreate(&streams[i]);   cudaMallocHost((void \*\*) &h\_a, size \* sizeof(float));  cudaMallocHost((void \*\*) &h\_b, size \* sizeof(float));  cudaMallocHost((void \*\*) &h\_c, size \* sizeof(float));  cudaMalloc((void \*\*)&d\_a, size \* sizeof(float));  cudaMalloc((void \*\*)&d\_b, size \* sizeof(float));  cudaMalloc((void \*\*)&d\_c, size \* sizeof(float));   for (int i = 0; i < size; i++) {  h\_a[i] = i;  h\_b[i] = i + 1;  }   for (int i = 0; i < stream\_num; i++) {  cudaMemcpyAsync(d\_a + i \* portion\_size, h\_a + i \* portion\_size, portion\_size \* sizeof(float), cudaMemcpyHostToDevice, streams[i]);  cudaMemcpyAsync(d\_b + i \* portion\_size, h\_b + i \* portion\_size, portion\_size \* sizeof(float), cudaMemcpyHostToDevice, streams[i]);  }  for (int i = 0; i < stream\_num; i++) cudaStreamSynchronize(streams[i]);   dim3 threadsPerBlock(256);  dim3 numBlocks((portion\_size + threadsPerBlock.x - 1) / threadsPerBlock.x);   cudaEventRecord(start, nullptr);  for (int i = 0; i < stream\_num; i++)  addVectors<<<numBlocks, threadsPerBlock, 0, streams[i]>>>(d\_a + i \* portion\_size, d\_b + i \* portion\_size, d\_c + i \* portion\_size, portion\_size);   for (int i = 0; i < stream\_num; i++) {  cudaMemcpyAsync(h\_c + i \* portion\_size, d\_c + i \* portion\_size, portion\_size \* sizeof(float), cudaMemcpyDeviceToHost, streams[i]);  }  for (int i = 0; i < stream\_num; i++) cudaStreamSynchronize(streams[i]);   cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);  cudaEventElapsedTime(&time, start, stop);  cout << "time = "<< time << endl;   //show\_mass(h\_c, 100);   cudaFree(d\_a);  cudaFree(d\_b);  cudaFree(d\_c);  cudaFree(h\_a);  cudaFree(h\_b);  cudaFree(h\_c);  for (int i = 0; i < stream\_num; i++) cudaStreamDestroy(streams[i]);   return 0; } |

Листинг 1 – программа LR09\_G2.cu

Результат работы программы:

|  |
| --- |
| time = 5.97504  Process finished with exit code 0 |



В результате программы мы видим с увеличением количеством потоков время работы увеличивается, для размера вектора в 32\*2^12 оптимально использовать 2 потока

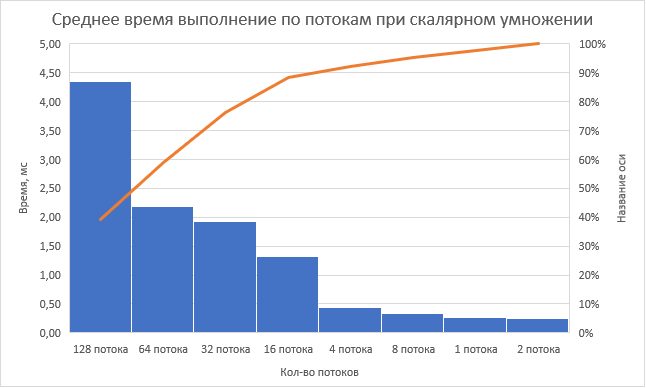
Для проверки скалярного умножение была взята таже программа но добавлена функция скалярного умножения dotProduct.

|  |
| --- |
| #include <iostream> #include <cuda.h> #include <cuda\_runtime.h>  using namespace std;  void show\_mass(float \*a, int num){  for (int i = 0; i < num; i++) {  printf("%f ",a[i]);  if (i%10 == 0) printf("\n");  }  printf("\n"); } **\_\_global\_\_** void dotProduct(float \*a, float \*b, float \*c, int n) {  **\_\_shared\_\_** float temp[256];  int index = blockIdx.x \* blockDim.x + threadIdx.x;  int stride = blockDim.x \* gridDim.x;  int tid = threadIdx.x;   temp[tid] = 0;  for (int i = index; i < n; i += stride) temp[tid] += a[i] \* b[i];   \_\_syncthreads();   int i = blockDim.x / 2;  while (i != 0) {  if (tid < i) temp[tid] += temp[tid + i];  \_\_syncthreads();  i /= 2;  }  if (tid == 0) c[blockIdx.x] = temp[0];  }  int main() {  int stream\_num = 128;  int num = 1 << 12;  int size = 32 \* num;  int portion\_size = size / stream\_num;   float \*h\_a, \*h\_b, \*h\_c;  float \*d\_a, \*d\_b, \*d\_c;  float time = 0;  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);   auto \*streams = (cudaStream\_t\*)calloc(stream\_num, sizeof(cudaStream\_t));  for (int i = 0; i < stream\_num; i++) cudaStreamCreate(&streams[i]);   cudaMallocHost((void \*\*) &h\_a, size \* sizeof(float));  cudaMallocHost((void \*\*) &h\_b, size \* sizeof(float));  cudaMallocHost((void \*\*) &h\_c, size \* sizeof(float));  cudaMalloc((void \*\*)&d\_a, size \* sizeof(float));  cudaMalloc((void \*\*)&d\_b, size \* sizeof(float));  cudaMalloc((void \*\*)&d\_c, size \* sizeof(float));   for (int i = 0; i < size; i++) {  h\_a[i] = i;  h\_b[i] = i + 1;  }   for (int i = 0; i < stream\_num; i++) {  cudaMemcpyAsync(d\_a + i \* portion\_size, h\_a + i \* portion\_size, portion\_size \* sizeof(float), cudaMemcpyHostToDevice, streams[i]);  cudaMemcpyAsync(d\_b + i \* portion\_size, h\_b + i \* portion\_size, portion\_size \* sizeof(float), cudaMemcpyHostToDevice, streams[i]);  }  for (int i = 0; i < stream\_num; i++) cudaStreamSynchronize(streams[i]);   dim3 threadsPerBlock(256);  dim3 numBlocks((portion\_size + threadsPerBlock.x - 1) / threadsPerBlock.x);   cudaEventRecord(start, nullptr);  for (int i = 0; i < stream\_num; i++)  dotProduct<<<numBlocks, threadsPerBlock, 0, streams[i]>>>(d\_a + i \* portion\_size, d\_b + i \* portion\_size, d\_c + i \* portion\_size, portion\_size);   for (int i = 0; i < stream\_num; i++) {  cudaMemcpyAsync(h\_c + i \* portion\_size, d\_c + i \* portion\_size, portion\_size \* sizeof(float), cudaMemcpyDeviceToHost, streams[i]);  }  for (int i = 0; i < stream\_num; i++) cudaStreamSynchronize(streams[i]);   cudaEventRecord(stop, nullptr);  cudaEventSynchronize(stop);  cudaEventElapsedTime(&time, start, stop);  cout << "time = "<< time << endl;   //show\_mass(h\_c, 100);   cudaFree(d\_a);  cudaFree(d\_b);  cudaFree(d\_c);  cudaFree(h\_a);  cudaFree(h\_b);  cudaFree(h\_c);  for (int i = 0; i < stream\_num; i++) cudaStreamDestroy(streams[i]);   return 0; } |

Листинг 1 – программа LR09\_G3.cu

Результат работы программы:

|  |
| --- |
| time = 6.53005  Process finished with exit code 0 |



При умножении также наблюдаем что при увелечении количества потоков время выполения увеличиваеться. Также видно что при использовании 2 потоков время выполнения меньше всего.

Вывод: в ходе выполнения лабораторной работы, была исследована и применена работа с библиотекой с закреплённой память, стало понятно что работа с ней быстрее по сравнению с обычным выделение памяти, также была исследована работа с cudastream, можно сделать вывод что при использовании одной видеокарты не имеет смысл использовать cudastream так с увеличением количества потоков время выполнения растет.