МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №*2***

**по курсу «Параллельная обработка данных»**

***Сортировка чисел на GPU. Свертка, сканирование, гистограмма.***

Выполнил: *А. Ю. Голов*

Группа: *М8О-401*

Преподаватель: А.Ю. Морозов

Москва, 2025

**Условие**

Цель работы. Ознакомление с фундаментальными алгоритмами GPU: свертка

(reduce), сканирование (blelloch scan) и гистограмма (histogram). Реализация одной из

сортировок на CUDA. Использование разделяемой и других видов памяти.

Исследование производительности программы с помощью утилиты nvprof.

**Программное и аппаратное обеспечение**

Видеокарта: NVIDIA GeForce RTX 2060 Mobile

* **Compute Capability**: 7.5
* **Графическая память**: 6 ГБ GDDR6, с 192-битной шиной и пропускной способностью 336 ГБ/с
* **Разделяемая память**: до 64 КБ на мультипроцессор
* **Константная память**: 64 КБ
* **Количество регистров на блок**: 65 536
* **Максимальное количество блоков на мультипроцессор**: 16
* **Максимальное количество нитей на мультипроцессор**: 1 024
* **Количество мультипроцессоров (SM)**: 30

Процессор: **Intel Core i7-9750H** имеет следующие характеристики:

* **Количество ядер**: 6
* **Количество потоков**: 12
* **Техпроцесс**: 14 нм

Оперативная память:

* Объём: 16 ГБ
* **Тактовая частота:** 3500 MHz
* **Поколение**: DDR4

Жёсткий диск:

* **Объём: 512 ГБ**
* Формат: SSD M2

Программное обеспечение:

* **Операционная система: Ubuntu 24.04 LTS**
* IDE: Lunar Vim

**Метод решения**

Методика решения основана на комбинации битонической сортировки и сортировки подсчётом. Сначала выполняется битоническая сортировка, разбивая массив на блоки и упорядочивая их параллельно. Затем с помощью чередующихся сравнений и обменов формируется битоническая последовательность, а после — глобальное слияние отсортированных частей.

После предварительной сортировки применяется сортировка подсчётом. Потоки параллельно вычисляют частоту элементов, затем определяют их финальные позиции с помощью префиксной суммы и записывают отсортированные значения в итоговый массив.

Для эффективной работы алгоритма данные распределяются по блокам, минимизируя конфликты доступа. Используется синхронизация на критических этапах, а нагрузка распределяется динамически в зависимости от размера массива.

**Описание программы**

\_\_device\_\_ void MergeStep(int\* data, int size, int left, int right, int step, int posX)

{

\_\_shared\_\_ int sharedMem[SIZE\_OF\_BLOCKS];

int\* tmp = data;

for (int i = left; i < right; i += step)

{

int idx;

tmp = data + i;

if (posX >= SIZE\_OF\_BLOCKS / 2) {

idx = SIZE\_OF\_BLOCKS \* 3 / 2 - 1 - posX;

}

else {

idx = posX;

}

if (posX >= SIZE\_OF\_BLOCKS / 2) {

sharedMem[posX] = tmp[idx];

}

else {

sharedMem[posX] = tmp[posX];

}

\_\_syncthreads();

for (int j = SIZE\_OF\_BLOCKS / 2; j > 0; j /= 2)

{

unsigned int XOR = posX ^ j;

if (XOR > posX)

{

if ((posX & SIZE\_OF\_BLOCKS) != 0)

{

if (sharedMem[posX] < sharedMem[XOR]){

thrust::swap(sharedMem[posX], sharedMem[XOR]);

}

}

else

{

if (sharedMem[posX] > sharedMem[XOR]){

thrust::swap(sharedMem[posX], sharedMem[XOR]);

}

}

}

\_\_syncthreads();

}

tmp[posX] = sharedMem[posX];

}

}

Выполняет один шаг битонического слияния. Загружает элементы в **разделяемую память**, упорядочивает их с помощью итеративных сравнений и обменов, затем записывает результат обратно.

\_\_global\_\_ void BitonicMerge(int\* data, int size, bool isOdd)

{

unsigned int posX = threadIdx.x;

int blockId = blockIdx.x;

int shift = gridDim.x;

if (isOdd) {

MergeStep(data, size, (SIZE\_OF\_BLOCKS / 2) + blockId \* SIZE\_OF\_BLOCKS, size - SIZE\_OF\_BLOCKS, shift \* SIZE\_OF\_BLOCKS, posX);

}

else {

MergeStep(data, size, blockId \* SIZE\_OF\_BLOCKS, size, shift \* SIZE\_OF\_BLOCKS, posX);

}

}  
Запускает MergeStep для параллельного слияния блоков. Определяет, какие части массива обрабатываются в текущем шаге, учитывая размер сетки.

\_\_global\_\_ void SortStep(int\* data, int j, int k, int size)

{

\_\_shared\_\_ int sharedMem[SIZE\_OF\_BLOCKS];

int\* tmp = data;

unsigned int posX = threadIdx.x;

int blockId = blockIdx.x;

int shift = gridDim.x;

for (int i = blockId \* SIZE\_OF\_BLOCKS; i < size; i += shift \* SIZE\_OF\_BLOCKS)

{

tmp = data + i;

sharedMem[posX] = tmp[posX];

\_\_syncthreads();

for (j = k / 2; j > 0; j /= 2)

{

unsigned int XOR = posX ^ j;

if (XOR > posX)

{

if ((posX & k) != 0)

{

if (sharedMem[posX] < sharedMem[XOR]){

thrust::swap(sharedMem[posX], sharedMem[XOR]);

}

}

else

{

if (sharedMem[posX] > sharedMem[XOR]){

thrust::swap(sharedMem[posX], sharedMem[XOR]);

}

}

}

\_\_syncthreads();

tmp[posX] = sharedMem[posX];

}

}

}  
Реализует битоническую сортировку внутри блоков. Копирует элементы в **разделяемую память**, выполняет сортировку с чередованием направлений, затем записывает результат обратно.

void BitonicSort(int\* devData, int partitionSize)

{

for (int i = 2; i <= partitionSize; i \*= 2)

{

if (i > SIZE\_OF\_BLOCKS){

break;

}

for (int j = i / 2; j > 0; j /= 2)

{

SortStep <<<NUM\_OF\_BLOCKS, SIZE\_OF\_BLOCKS>>> (devData, j, i, partitionSize);

CSC(cudaGetLastError());

}

}

for (int i = 0; i < 2 \* (partitionSize / SIZE\_OF\_BLOCKS); ++i)

{

BitonicMerge <<<NUM\_OF\_BLOCKS, SIZE\_OF\_BLOCKS>>> (devData, partitionSize, (i % 2 == 0));

CSC(cudaGetLastError());

}

}  
Запускает битоническую сортировку и последовательное слияние блоков. Поэтапно увеличивает размер сортируемых частей, затем объединяет отсортированные фрагменты.

**Результаты**

Результаты замеров производительности программ в зависимости от размера сортируемого массива. Данные представлены в секундах.

|  |  |  |
| --- | --- | --- |
| Размер массива | GPU + Parallel | CPU |
| 1000 | 0.00054 | 0.000555764 |
| 10000 | 0.000652603 | 0.0107603 |
| 100000 | 0.0109739 | 0.0951744 |
| 1000000 | 0.37084 | 1.22396 |

Ниже приведены выводы утилиты nvprof при сортировке массива из 10 элементов.

➜ lab5 git:(main) ✗ nvprof ./a.out

* + - 1. ==7147== NVPROF is profiling process 7147, command: ./a.out
      2. Sorting 10 elements took 0.00750193 seconds
      3. ==7147== Profiling application: ./a.out
      4. ==7147== Profiling result:
      5. Type Time(%) Time Calls Avg Min Max Name
      6. GPU activities: 87.21% 15.935us 6 2.6550us 2.3680us 2.9430us SortStep(int\*, int, int, int)
      7. 7.71% 1.4080us 1 1.4080us 1.4080us 1.4080us [CUDA memcpy DtoH]
      8. 5.08% 928ns 1 928ns 928ns 928ns [CUDA memcpy HtoD]
      9. API calls: 92.73% 171.13ms 1 171.13ms 171.13ms 171.13ms cudaMalloc
      10. 3.96% 7.3002ms 6 1.2167ms 2.2580us 7.2851ms cudaLaunchKernel
      11. 3.13% 5.7749ms 114 50.656us 127ns 3.2898ms cuDeviceGetAttribute
      12. 0.12% 217.80us 2 108.90us 23.724us 194.08us cudaMemcpy
      13. 0.05% 89.428us 1 89.428us 89.428us 89.428us cudaFree
      14. 0.01% 21.183us 1 21.183us 21.183us 21.183us cuDeviceGetName
      15. 0.00% 4.9770us 1 4.9770us 4.9770us 4.9770us cuDeviceGetPCIBusId
      16. 0.00% 1.8670us 3 622ns 152ns 1.5000us cuDeviceGetCount
      17. 0.00% 1.7260us 2 863ns 130ns 1.5960us cuDeviceGet
      18. 0.00% 677ns 6 112ns 60ns 346ns cudaGetLastError
      19. 0.00% 617ns 1 617ns 617ns 617ns cuDeviceTotalMem
      20. 0.00% 310ns 1 310ns 310ns 310ns cuModuleGetLoadingMode
      21. 0.00% 283ns 1 283ns 283ns 283ns cuDeviceGetUuid

➜ lab5 git:(main) ✗ nvprof --print-gpu-trace ./a.out

==7220== NVPROF is profiling process 7220, command: ./a.out

Sorting 10 elements took 0.000424398 seconds

==7220== Profiling application: ./a.out

==7220== Profiling result:

Start Duration Grid Size Block Size Regs\* SSMem\* DSMem\* Size Throughput SrcMemType DstMemType Device Context Stream Name

3.17090s 928ns - - - - - 40B 41.107MB/s Pageable Device NVIDIA GeForce 1 7 [CUDA memcpy HtoD]

3.17122s 3.0400us (16 1 1) (1024 1 1) 16 4.0000KB 0B - - - - NVIDIA GeForce 1 7 SortStep(int\*, int, int, int) [127]

3.17123s 2.7840us (16 1 1) (1024 1 1) 16 4.0000KB 0B - - - - NVIDIA GeForce 1 7 SortStep(int\*, int, int, int) [129]

3.17123s 2.4630us (16 1 1) (1024 1 1) 16 4.0000KB 0B - - - - NVIDIA GeForce 1 7 SortStep(int\*, int, int, int) [131]

3.17123s 2.6240us (16 1 1) (1024 1 1) 16 4.0000KB 0B - - - - NVIDIA GeForce 1 7 SortStep(int\*, int, int, int) [133]

3.17124s 2.6240us (16 1 1) (1024 1 1) 16 4.0000KB 0B - - - - NVIDIA GeForce 1 7 SortStep(int\*, int, int, int) [135]

3.17124s 2.6240us (16 1 1) (1024 1 1) 16 4.0000KB 0B - - - - NVIDIA GeForce 1 7 SortStep(int\*, int, int, int) [137]

3.17125s 1.4080us - - - - - 40B 27.093MB/s Device Pageable NVIDIA GeForce 1 7 [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.

SSMem: Static shared memory allocated per CUDA block.

DSMem: Dynamic shared memory allocated per CUDA block.

SrcMemType: The type of source memory accessed by memory operation/copy

DstMemType: The type of destination memory accessed by memory operation/copy

**Выводы**

**Область применения**  
Битоническая сортировка предназначена для **параллельных вычислений** и эффективна в условиях GPU, где важно минимизировать зависимости между потоками. Она используется в задачах обработки больших массивов данных, в том числе в высокопроизводительных вычислениях, финансовом анализе и машинном обучении.

**Типовые задачи**  
Этот алгоритм полезен в **реальном времени** при обработке данных на GPU, например, в **графике, моделировании физических процессов и финансовых вычислениях**. Он обеспечивает детерминированную сортировку, что важно в системах с жесткими временными ограничениями.

**Сложность программирования и возникшие проблемы**  
Реализация битонической сортировки на **GPU сложнее**, чем на CPU, из-за необходимости работы с **разделяемой памятью**, синхронизации потоков и оптимизации доступа к памяти. При разработке возникли ошибки **выхода за границы массива** и **некорректного обращения к памяти GPU**, которые потребовали дополнительной обработки.

**Сравнение и объяснение результатов**  
Замеры показывают, что **GPU быстрее CPU** на больших массивах, но на маленьких входных данных задержка вызова ядер CUDA и операций cudaMemcpy нивелирует преимущество параллельных вычислений.

* **Для 1000 элементов** разница между CPU и GPU минимальна, так как накладные расходы CUDA слишком велики.
* **Для 10⁶ элементов** GPU оказывается почти в **3 раза быстрее**, демонстрируя эффективность параллелизма.
* **Выводы nvprof** показывают, что основное время работы уходит на выполнение SortStep, а затраты на cudaMemcpy становятся значимыми при малых данных.

В целом, **GPU-версия превосходит CPU при больших объемах данных**, но на малых массивах оправданнее использовать CPU-реализацию из-за меньших накладных расходов.