Нижегородский государственный университет им. Н.И. Лобачевского

Факультет Вычислительной математики и кибернетики

Рекомендации по оптимизации, использование библиотек CUDA

Бастраков С.И.

ВМК ННГУ

bastrakov@vmk.unn.ru

Молодежная научная школа «Суперкомпьютерные технологии и высокопроизводительные вычисления в образовании, науке и промышленности» 5 – 8 ноября 2014 г., г. Нижний Новгород

Содержание

- Рекомендации по оптимизации
- Использование CUBLAS
- Использование CUFFT
- Использование CURAND

Общие рекомендации

Выбор алгоритма

- Выбор подходящих для параллельных вычислений алгоритмов. Типичные схемы декомпозиции задачи:
  - разбиение на большое число (тысячи и более) независимых подзадач;
  - разбиение на относительно небольшое число независимых подзадач, каждая из которых делится на относительно большое число подзадач.
- Предпочтительны алгоритмы с большой долей вычислений по отношению к количеству обращений к памяти.
- Использование CPU для последовательных вычислений и GPU для параллельных. Минимизация обмена между хостом и устройством и/или выполнение его асинхронно.

Асинхронное выполнение операций

- Для асинхронного выполнения операций (копирование данных, исполнения ядер) и использования нескольких устройств используется механизм потоков cudaStream.
  - Не путать с потоками, исполняющими ядра.
- cudaStreamCreate – создание потока.
- cudaStreamDestroy – уничтожение потока.
- cudaStreamSynchronize – ожидание завершения всех операций потока.
- cudaDeviceSynchronize – ожидание завершения всех операций всех потоков.

Пример

- Создание 2 потоков и выделение памяти в специальной области, допускающей асинхронное копирование:

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Пример

- Асинхронный запуск копирования данных и запуск ядра:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
   MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Пример

- Ожидание завершения операций потоков:

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Оптимизация работы с памятью

- Основной вид оптимизации приложений на GPU – оптимизация работы с памятью.
- Использование эффективных паттернов доступа к глобальной и разделяемой памяти.
- Хранение многократно используемых данных в кэше/разделяемой памяти.

Оптимизация загрузки устройства

- Число потоков на мультипроцессор обычно гораздо больше числа CUDA-ядер.
- Большое число потоков обеспечивает скрытие латентности доступа к памяти, особенно глобальной.
- **Степень загрузки устройства** (*occupancy*) – отношение количества варпов, работающих на мультипроцессоре, к максимально возможному количеству варпов.
- Ограничивающим для числа потоков фактором является количество разделяемой памяти и регистров.
- При наличии достаточного количества ресурсов несколько блоков могут одновременно исполняться на одном мультипроцессоре.

Оптимизация арифметических инструкций

- Рекомендуется использовать везде, где это допустимо по точности, арифметику с плавающей запятой одинарной точности.
- На устройстве доступны быстрые и менее точные варианты трансцендентных функций: __sinf(x), __cosf(x), __expf(x) и др., ключ компилятора -use_fast_math заменяет все вызовы трансцендентных функций на устройстве на быстрые варианты.

Использование инструментов и библиотек

- Использование оптимизированных библиотек в составе CUDA Toolkit: CUBLAS, CUFFT, CURAND, CUSPARSE, NPP, Thrust.
  - Имеются также сторонние библиотеки, в том числе открытые.
- Использование профилировщика для определения «узких мест» и получения рекомендаций по увеличению производительности.
- Опция компилятора --ptxas-options = -v для получения сведений о количестве используемой памяти всех видов.
  - Возможность для определения оптимального числа потоков в блоке на основании данных устройства и информации о количестве используемой памяти.

Использование CUBLAS

BLAS

- BLAS (Basic Linear Algebra Subprograms) – набор базисных подпрограмм линейной алгебры.
- Данный набор является основой для функций из пакета LAPACK.
- Впервые опубликован в 1979.
- Множество оптимизированных реализаций на Fortran и C, в том числе от производителей устройств:
  - Goto BLAS
  - BLAS в составе Intel MKL
  - BLAS в составе ACML
  - CUBLAS в составе CUDA Toolkit с версии 1.0
  - …

Структура BLAS

- BLAS состоит из 3 уровней:
  - операции над векторами (vector-vector);
  - вектор-матричные операции (matrix-vector);
  - операции над матрицами (matrix-matrix).

Соглашения об именовании

- Имя любой процедуры BLAS имеет следующую структуру:

  <character code><name><mode>(…)

- Имя функции CUBLAS, соответствующей функции func, имеет вид cublasFunc.

Соглашения об именовании

- <character code> – символ, описывающий тип данных, с которым работает процедура:

s	вещественный, одинарной точности
c	комплексный, одинарной точности
d	вещественный, двойной точности
z	комплексный, двойной точности

- Для некоторых процедур и функций данные символы могут комбинироваться. Например, функция *scasum* принимает на вход массив комплексных чисел и возвращает вещественное значение.

Соглашения об именовании

- <name>: для BLAS Level 1 определяет тип операции (например, *dot* – скалярное произведение, *swap* – перестановка элементов векторов местами), для BLAS Level 2 и 3 определяет тип матричного аргумента.
- <mode>: определяет дополнительный тип операции, например, размерность.

Индексирование

- BLAS использует представление данных в стиле Fortran: хранение матриц по столбцам и индексирование (на логическом уровне) с 1.
- Некоторые функции BLAS возвращают индекс элемента массива. Независимо от того, какая версия библиотеки (Fortran или C) используется, элементы массива нумеруются с 1. Следовательно, при использовании C-версии из результата, который вернула функция, следует вычесть 1.

Использование CUBLAS

- Общая концепция: вызов функций (не ядер) из **cublas.h** на стороне хоста, используется **cublas.lib**.
- Данные копируются в память устройства при помощи специальных функций.
- Все функции возвращают статус, по которому можно проверять ошибки.
- **cublasInit()** - инициализация CUBLAS (должна быть вызвана перед использованием любой другой CUBLAS функции).
- **cublasShutdown()** - освобождает ресурсы, используемые библиотекой CUBLAS.

Использование CUBLAS

- **cublasAlloc**(int n, int elemSize, void **devicePtr) – создает объект в пространстве памяти GPU, содержащий массив из n элементов размера elemSize. Указатель на созданный объект размещается в devicePtr (данный указатель не должен впоследствии изменяться в коде, выполняемом на хосте)
- **cublasFree**(const void *devicePtr) – освобождает память, выделенную на GPU

Использование CUBLAS

- **cublasSetVector** (int n, int elemSize, const void *x, int incx, void *y, int incy) – копирует n элементов вектора x (пространство памяти CPU) в вектор y (пространство памяти GPU)
- **cublasGetVector** (int n, int elemSize, const void *x, int incx, void *y, int incy) – копирует n элементов вектора x (пространство памяти GPU) в вектор y (пространство памяти CPU)

Использование CUBLAS

- **cublasSetMatrix** (int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb) – копирует rows*cols элементов матрицы A (пространство памяти CPU) в матрицу B (пространство памяти GPU); матрицы хранятся по столбцам, lda – ведущая размерность матрицы A, ldb – ведущая размерность матрицы B
- **cublasGetMatrix** (int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb) – копирует rows*cols элементов матрицы A (пространство памяти CPU) в матрицу B (пространство памяти GPU); см. cublasSetMatrix

Некоторые функции CUBLAS

- void **cublasScopy** (int n, const float *x, int incx, float *y, int incy) – осуществляет копирование float-вектора x в вектор y, векторы x, y хранятся в глобальной памяти GPU со смещениями incx, incy
- float **cublasSdot** (int n, const float *x, int incx, const float *y, int incy) – вычисляет скалярное произведение float-векторов x и y длины n, хранящихся в глобальной памяти GPU со смещениями incx, incy
- void **cublasSaxpy** (int n, float alpha, const float *x, int incx, float *y, int incy) – выполняет операцию y += alpha * x для float-векторов x и y длины n, хранящихся в глобальной памяти GPU со смещениями incx, incy, и float-скаляра alpha

Некоторые функции CUBLAS

- void **cublasSgemv** (char trans, int m, int n, float alpha, const float *A, int lda, const float *x, int incx, float beta, float *y, int incy) – выполняет операцию y=alpha*op(A)*x + beta*y, где

op(A)=A при trans='N' или 'n', op(A)=A^T при trans='T', 't', 'C' или 'c', m – количество строк матрицы A, n – количество столбцов матрицы A, alpha, beta – float-скаляры, A – матрица, записанная по стоблцам, lda – ведущая размерность матрицы A, x, y – float-векторы длины n, хранящихся в глобальной памяти GPU со смещениями incx, incy

Использование CUFFT

FFT

- FFT – быстрое преобразование Фурье, быстрый алгоритм выполнения дискретного преобразования Фурье.
- Множество оптимизированных реализаций, в том числе от производителей устройств:
  - FFTW (независимая);
  - FFT в составе Intel MKL;
  - CUFFT в составе CUDA Toolkit с версии 1.0;
  - …

Функциональность CUFFT

- **1D**, **2D**, **3D** преобразования вещественных и комплексных данных с плавающей запятой одинарной и двойной точности.
- Возможность параллельного выполнения нескольких преобразований.
- **In-place**, **out-of-place** преобразования.
- Требования на длину входа в виде максимального значения:
  - 8 млн для 1D;
  - 16384 по каждой размерности для 2D и 3D.

Использование CUFFT

- Заголовочный файл **cufft.h**.
- Библиотека **cufft.lib**.
- Работа с CUFFT API на стороне хоста.
  - Все функции CUFFT API возвращают cufftResult.
- Данные и вычисления на GPU.
- Схема использования во многом напоминает FFTW.

Типы данных и типы преобразований

- Используются следующие типы данных:
  - **cufftReal** = float (сокращенно **R**);
  - **cufftDoubleReal** = double (сокращенно **D**);
  - **cufftComplex** = float2 (сокращенно **C**);
  - **cufftDoubleComplex** = double2 (сокращенно **Z**).
- Поддерживаются следующие типы преобразований: **CUFFT_R2C**, **CUFFT_C2R**, **CUFFT_C2C**, **CUFFT_D2Z**, **CUFFT_Z2D**, **CUFFT_Z2Z**.
- Направление преобразования определяется макросами **CUFFT_FORWARD** и **CUFFT_INVERSE**.

Создание и уничтожение плана

- Каждое преобразование описывается описывается **планом** (**plan**).
- План имеет тип cufftHandle.
- Для создания плана используются функции:

  cufftPlan1d(…), **cufftPlan2d**(…), **cufftPlan3d**(…), **cufftPlanMany**(…).
- cufftResult **cufftPlan1d**(cufftHandle* plan, int nx, cufftType type, int batch) – создает план одномерного преобразования типа type сигнала длины nx, batch равно количеству преобразований.
- Для уничтожения плана используется функция cufftResult **cufftDestroy**(cufftHandle plan).

Выполнение преобразования

- Выполнение преобразования соответствующего типа с заданным планом выполняется функциями **cufftExecC2C**(…), **cufftExecR2C**(…), **cufftExecC2R**(…), **cufftExecZ2Z**(…), **cufftExecD2Z**(…), **cufftExecZ2D**(…).
- cufftResult **cufftExecC2C**(cufftHandle plan, cufftComplex* idata, cufftComplex* odata, int direction) – выполняет преобразование над данными idata с заданным планом plan, результат записывается в odata, направление определяется direction:
  - idata, odata – указатели на device память, могут совпадать (в этом случае in-place преобразование).
  - При преобразованиях в обе стороны масштабирование не выполняется.

Пример

- Выполнение прямого и обратного одномерного комплексного преобразования одинарной точности
- BATCH=10 сигналов длины NX=256
- In-place преобразование
- После выполнения обратного преобразования данные отличаются от исходных умножением на NX

Пример

```
#define NX 256
#define BATCH 10
cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data,
    sizeof(cufftComplex)*NX*BATCH);
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);
cufftExecC2C(plan, data, data, CUFFT_FORWARD);
cufftExecC2C(plan, data, data, CUFFT_INVERSE);
cufftDestroy(plan);
cudaFree(data);
```

Использование CURAND

Псевдо- и квазислучайные числа

- Псевдослучайная последовательность генерируется детерминированным алгоритмом, но обладает большинством статистических свойств истинно случайной последовательности.
- Квазислучайная последовательность генерируется детерминированным алгоритмом и равномерно заполняет некоторый объем (но при этом порядок следования может быть «далек от случайного»).
- Частое использование при моделировании процессов при помощи метода Монте-Карло (например, в финансовой математике).

Структура CURAND

- CURAND – оптимизированная библиотека для генерирования псевдо- и квазислучайных чисел на хосте и GPU.
- Появилась в августе 2010.
- Состоит из 2 частей:
  - Host API: библиотека функций, вызываемых со стороны хоста, генерация может осуществляться как на хосте, так и на GPU.
  - Device API: набор device-функций для вызовов из ядер.

Host API

- Заголовочный файл **curand.h**, библиотека **curand.lib**.
- **curandCreateGenerator/curandCreateGeneratorHost** (curandGenerator_t * generator, curandRngType_t rng_type) – создает генератор заданного типа на GPU/хосте
  - В настоящее поддерживаются только 2 типа генераторов.
- **curandSetPseudoRandomGeneratorSeed** (curandGenerator_t generator, unsigned long long seed) – задает сид генератора.
- **curandDestroyGenerator** (curandGenerator_t generator) – освобождает ресурсы.

Host API

- **curandGenerate** (curandGenerator_t generator, unsigned int * outputPtr, size_t num) – осуществляет генерацию num чисел со всеми случайными битами, записывая их в outputPtr (память на хосте/девайсе должна быть выделена заранее).
- **curandGenerateNormal** (curandGenerator_t generator, float * outputPtr, size_t n, float mean, float stddev) – осуществляет генерацию num нормально распределенных чисел с заданными параметрами, записывая их в outputPtr (память на хосте/девайсе должна быть выделена заранее).

Host API

- **curandGenerateUniform** (curandGenerator_t generator, float * outputPtr, size_t num) – осуществляет генерацию num равномерно распределенных на (0,1] чисел, записывая их в outputPtr (память на хосте/девайсе должна быть выделена заранее).
- **curandGenerateUniformDouble** (curandGenerator_t generator, float * outputPtr, size_t num) – аналог curandGenerateUniform для чисел типа double.

Пример использования Host API

- Данный пример взят из документа CURAND Library, включенного в CUDA Toolkit, макросы для контроля ошибок удалены для краткости.

```
/*
* This program uses the host CURAND API to generate100
* pseudorandom floats.
*/
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>
```

Пример использования Host API

```
int main(int argc, char *argv[])
{
    size_t n = 100;
    size_t i;
    curandGenerator_t gen;
    float *devData, *hostData;
    /* Allocate n floats on host */
    hostData = (float *)calloc(n, sizeof(float));
    /* Allocate n floats on device */
    cudaMalloc((void **)&devData, n * sizeof(float));
```

Пример использования Host API

```
/* Create pseudo-random number generator */
    curandCreateGenerator(&gen,
          CURAND_RNG_PSEUDO_DEFAULT);
    /* Set seed */
    curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
    /* Generate n floats on device */
    curandGenerateUniform(gen, devData, n);
    /* Copy device memory to host */
    cudaMemcpy(hostData, devData, n * sizeof(float),
          cudaMemcpyDeviceToHost);
```

Пример использования Host API

```
    /* Show result */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");
    /* Cleanup */
    curandDestroyGenerator(gen);
    cudaFree(devData);
    free(hostData);
    return EXIT_SUCCESS;
}
```

Device API

- Интерфейс в **curand_kernel.h**.
- Все функции __device__ (т.е. вызываются из ядер).
- Работа осуществляется через состояния типа curandState* и curandStateSobol32*.
- Необходимо правильно инициализировать состояния для получения чисел из одной последовательности / чисел из некоррелированных последовательностей.

Device API: псевдослучайные числа

- __device__ void **curand_init** (unsigned long long seed, unsigned long long sequence, unsigned long long offset, curandState *state) – устанавливает state с заданными параметрами:
  - Разным значениям seed соответствуют разные состояния.
  - Период равен $2^{67} \cdot$ sequence + offset.
  - Последовательности с разными seed обычно (но не всегда) некоррелированы.
  - Последовательности с одинаковыми seed и разными sequence всегда некоррелированы.

Device API: псевдослучайные числа

- __device__ unsigned int **curand** (curandState *state) – возвращает следующий член последовательности псевдослучайных чисел, обновляет state
- __device__ float **curand_uniform** (curandState *state) – возвращает следующий член последовательности равномерно распределенных на (0, 1] псевдослучайных чисел, обновляет state

Device API: квазислучайные числа

- __device__ void **curand_init** (unsigned int *direction_vectors, unsigned int offset, curandStateSobol32 *state) – устанавливает state с заданными параметрами:
  - При любых параметрах период равен 2^{32}.
- __device__ unsigned int **curand** (curandStateSobol32 *state) – аналогично с curand (curandState *state).
- __device__ float **curand_uniform** (curandStateSobol32 *state) – аналогично с curand_uniform (curandState *state).

Пример использования Device API

- Данный пример взят из документа CURAND Library, включенного в CUDA Toolkit, макросы для контроля ошибок удалены для краткости.

```
/*
 * This program uses the device CURAND API to calculate
 * what proportion of pseudo-random ints have low bit set.
 */
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>
```

Пример использования Device API

```
__global__ void setup_kernel(curandState *state)
{
    int id = threadIdx.x + blockIdx.x * 64;
    /* Each thread gets same seed, a different sequence
        number, no offset */
    curand_init(1234, id, 0, &state[id]);
}
```

Пример использования Device API

```
__global__ void generate_kernel(curandState *state, int *result)
{
    int id = threadIdx.x + blockIdx.x * 64;
    int count = 0;
    unsigned int x;
    /* Copy state to local memory for efficiency */
    curandState localState = state[id];
```

Пример использования Device API

```
    /* Generate pseudo-random unsigned ints */
    for(int n = 0; n < 100000; n++) {
        x = curand(&localState);
        /* Check if low bit set */
        If(x & 1) { count++; }
    }
    /* Copy state back to global memory */
    state[id] = localState;
    /* Store results */
    result[id] += count;
}
```

Пример использования Device API

```
int main(int argc, char *argv[])
{
    int i, total;
    curandState *devStates;
    int *devResults, *hostResults;
    /* Allocate space for results on host */
    hostResults = (int *)calloc(64 * 64, sizeof(int));
    /* Allocate space for results on device */
    cudaMalloc((void **)&devResults, 64 * 64 * sizeof(int));
    /* Set results to 0 */
    cudaMemset(devResults, 0, 64 * 64 * sizeof(int));
```

Пример использования Device API

```
/* Allocate space for prng states on device */
cudaMalloc((void **)&devStates, 64 * 64 *
        sizeof(curandState));
/* Setup prng states */
setup_kernel<<<64, 64>>>(devStates);
/* Generate and use pseudo-random */
for(i = 0; i < 10; i++) {
        generate_kernel<<<64, 64>>>(devStates,
        devResults);
}
```

Пример использования Device API

```
/* Copy device memory to host */
    cudaMemcpy(hostResults, devResults, 64 * 64 *
            sizeof(int), cudaMemcpyDeviceToHost);
    /* Show result */
    total = 0;
    for(i = 0; i < 64 * 64; i++) {
            total += hostResults[i];
    }
    printf("Fraction with low bit set was %10.13f\n",
    (float)total / (64.0f * 64.0f * 100000.0f * 10.0f));
```

Пример использования Device API

```
    /* Cleanup */
    cudaFree(devStates);
    cudaFree(devResults);
    free(hostResults);
    return EXIT_SUCCESS;
}
```

Материалы

- NVIDIA CUDA C Programming Guide
- Материалы курса по CUDA и OpenACC: http://www.nvidia.ru/object/cuda-openacc-online-course-ru.html
- Боресков А.В. и др. «Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие»
- Сандерс Дж., Кэндрот Э. «Технология CUDA в примерах: введение в программирование графических процессоров»
- Farber R. CUDA Application Design and Development