Проект комиссии Президента
по модернизации и технологическому развитию экономики России
«Создание системы подготовки высококвалифицированных кадров
в области суперкомпьютерных технологий и
специализированного программного обеспечения»

УТВЕРЖДАЮ
Председатель экспертного совета
системы НОЦ СКТ, член-корр. РАН
В.В. Воеводин
_____
"_____"_____201__г.

Конспект лекций дисциплины

«Параллельное программирование для многопроцессорных систем
с общей и распределенной памятью»

«**010400.62 – Прикладная математика и информатика**»

Разработчики: Лаева В.И., Трунов А.А.
Рецензент: проф. Вшивков В.А.

Москва

CUDA. Лекция № 2

Иерархия памяти

Каждый поток имеет доступ к регистрам – это самая быстрая разновидность памяти.

Каждый поток имеет собственную локальную память (local memory), физически находящуюся в глобальной памяти ГПУ. Переменные в локальной памяти не доступны другим потокам.

Каждый блок потоков имеет разделяемую память (shared memory), доступную потокам только в пределах блока, её размер составляет десятки кБ. Разделяемая память блока хранит данные только во время выполнения блока. При выполнении другого блока на том же мультипроцессоре содержимое разделяемой памяти может измениться.

Все потоки в пределах ГПУ имеют доступ к глобальной памяти (global memory), её размер составляет единицы - десятки ГБ.

Константная память (constant memory). Используется только для чтения и физически находится в глобальной памяти. Содержимое константной памяти кэшируется.

Текстурная память (texture memory). Предназначена для операций с текстурами для целей компьютерной графики. С текстурной памятью возможны только операции чтения. Физически данный вид памяти находится в глобальной памяти. Содержимое кэшируется.

Спецификаторы переменных

Спецификатор	Размещение	Примечание
<нет>	регистры или локальная память	на усмотрение компилятора
__device__	глобальная память	
__constant__	константная память	
__shared__	разделяемая память	
__restrict__	-	ограниченные указатели

Ограниченные указатели применяются для указания компилятору оптимизировать код, если в программе используются неперекрывающиеся массивы памяти (векторы, матрицы и др.).

Работа с глобальной памятью: эффективный доступ

Для эффективной работы с глобальной памятью необходимо соблюдать некоторые шаблоны:

- Важно объединять запросы к памяти от нескольких потоков в один запрос.
- Границы запрашиваемых из глобальной памяти данных должны быть выровнены (aligned) по 32-, 64- или 128-байтным сегментам памяти.
- Для архитектур предыдущих поколений (compute capability 1.x) важным является согласованный (coalesced) доступ: номер потока должен определённым образом быть согласован со смещением адреса.
- Важным для быстродействия является принцип локальности данных: потоки в пределах полуварпа/варпа должны обращаться к смежным адресам в глобальной памяти.
- Один из простых шаблонов: последовательное обращение к смежным адресам памяти в пределах варпа (warp).

При этом следует помнить, что традиционно матрицы в языках C/C++ хранятся "по строкам", т.е. по последовательным адресам памяти хранятся последовательные элементы строки. В то же время в языке Fortran матрицы хранятся "по столбцам", т.е. по последовательным адресам памяти хранятся последовательные элементы столбца.

Пример согласованного (coalesced) доступа к элементам вектора:

```
unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Пример несогласованного доступа к элементам вектора:

```
unsigned int i = (blockIdx.x * blockDim.x + threadIdx.x * 513)
% N;
```

где

```
N - размер вектора
blockDim.x = 32.
```

Работа с разделяемой памятью: кэширование

ГПУ с версией архитектуры < 2.0 (compute capability < 2.0) не кэшируют запросы к глобальной памяти. Однако при каждом запросе к глобальной памяти обычно передаётся блок данных существенно большего размера, чем запрашиваемый. При отсутствии кэш-памяти происходят потери немедленно неиспользованной памяти.

Поэтому для достижения высокой производительности часто приходится реализовывать ручное кэширование (повторное использование данных). Для кэширования в этом случае используется разделяемая память.

Типовой шаблон работы с разделяемой памятью:

- загрузить небольшую часть данных из глобальной памяти в разделяемую память блока;
- синхронизировать потоки в пределах блока;
- провести вычисления с данными из разделяемой памяти;

- сохранить результат в глобальной памяти.

Синхронизация осуществляется для того, чтобы все запрошенные данные были доступны всем потокам в пределах блока.

Работа с разделяемой памятью: конфликты доступа

Эффективный доступ к разделяемой памяти также имеет свой шаблон.

Разделяемая память разбита на 16/32 банков, каждый из банков способен выполнить одно чтение или запись 32-битового слова. К каждому из этих банков потоки полуварпа могут обратиться параллельно.

Если несколько потоков (но не все!) в пределах полуварпа обращаются к одному банку, то происходит конфликт доступа к банку.

В случае конфликта доступа обращение к банкам происходит последовательно, что снижает производительность.

Шаблон: обращение по последовательным адресам разделяемой памяти приводит к обращению к разным банкам.

Фрагмент программы: матрично-векторное произведение: версия 1.0.

```
__global__ void simpleMatVecMul (float *A, float *x, float *y)
{
     int row = blockIdx.x * blockDim.x + threadIdx.x;
     float sum = 0.0f;
     for (int i = 0; i < N; i++)
          sum += A[row*N + i] * x[i];
     y[row] = sum;
}
```

В алгоритме матрично-векторного произведения значение каждого элемента матрицы A используется однократно. При этом в приведённом простом фрагменте программы на каждую строку матрицы A производится копирование всех элементов вектора x. Таким образом, количество обращений к глобальной памяти составит: 2*N*N. Здесь и далее предполагается размерность матрицы A – [NxN], размер вектора x – [N]. Подсчёт приведён для системы, в которой отсутствует автоматическое кэширование. Для системы с кэшированием количество реальных чтений глобальной памяти будет существенно меньше.

Фрагмент программы: матрично-векторное произведение: версия 1.1 (non-coalesced, кэш-промахи).

```
__global__ void simpleMatVecMul (float *A, float *x, float *y)
{
int row = (blockIdx.x * blockDim.x + 513 * threadIdx.x) % N;
float sum = 0.0f;
for (int i = 0; i < N; i++)
     sum += A[row*N + i] * x[i];
y[row] = sum;
```

```
}
где N = 16000
blockDim.x = 32
```
Данный фрагмент на системах с автоматическим кэшированием выполняется в несколько раз медленнее, чем исходный фрагмент.

Фрагмент программы: матрично-векторное произведение: версия 2 (повторное использование вектора x).

```
__global__ void shared_x_MatVecMul (float *A, float *x, float
*y)
{
     int row = blockIdx.x * blockDim.x + threadIdx.x;
     float sum = 0.0f;
     __shared__ float x_tile[BLOCK_SIZE];

     for (int i = 0; i < N/BLOCK_SIZE; i++) {
          x_tile[threadIdx.x] = x[i*BLOCK_SIZE + threadIdx.x];
          __syncthreads();
          for (int j = 0; j < BLOCK_SIZE; j++)
               sum += A[row*N + i*BLOCK_SIZE + j] * x_tile[j];
     }
     y[row] = sum;
}
```

Данная версия программы выполняется с такой же скоростью, как и версия 1.0 на системах с автоматическим кэшированием и в несколько раз быстрее на системах без кэш-памяти.

Фрагмент программы: матрично-векторное произведение: версия 3 (добавлено блочное копирование матрицы A).

```
__global__ void shared_A_x_MatVecMul (float *A, float *x,
float *y)
{
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  __shared__ float x_tile[BLOCK_SIZE];
  __shared__ float A_tile[BLOCK_SIZE][BLOCK_SIZE];

  for (int i = 0; i < N/BLOCK_SIZE; i++) {
    x_tile[threadIdx.x] = x[i*BLOCK_SIZE + threadIdx.x];
    for (int k = 0; k < BLOCK_SIZE; k++)
      A_tile[k][threadIdx.x]=
      A[(blockIdx.x*blockDim.x + k)*N + i*BLOCK_SIZE +
threadIdx.x];
      __syncthreads();
      for (int j = 0; j < BLOCK_SIZE; j++)
         sum += A_tile[threadIdx.x][j] * x_tile[j];
  }
  y[row] = sum;
}
```

Данная версия выполняется до нескольких раз быстрее на системах с автоматическим кэшированием и до 10-20 раз быстрее на системах без кэш-памяти. Сравнение делалось относительно версии 1.0.

Ветвления

В силу аппаратных особенностей все потоки в пределах варпа выполняют одну инструкцию (SIMD / SIMT архитектура).

При выполнении условных конструкций каждый поток в пределах варпа выполняет ветку if и ветку else, что уменьшает производительность. Если для данного потока инструкция не должна быть выполнена, то результаты его работы не сохраняются.

Для нормальной работоспособности вышеприведённые примеры программ должны быть снабжены условными конструкциями, предупреждающими некорректные операции с памятью, если размер задачи не кратен количеству потоков в блоке.

Модифицированные примеры заметно теряют в производительности.