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 № 3

Синхронизация: внутри блока потоков

Барьерная синхронизация потоков в пределах блока осуществляется вызовом функции **void __syncthreads()**. Барьер можно применять в условных конструкциях, но при условии, что все потоки блока достигнут его. Барьер не влияет на потоки из других блоков, т.е. осуществляет локальную синхронизацию.

ГПУ архитектуры (compute capability) 2.x поддерживают ещё 3 варианта барьерной синхронизации:

- **`int __syncthreads_count(int predicate)`** - барьер и подсчёт кол-ва потоков блока, для которых predicat != 0;
- **`int __syncthreads_and(int predicate)`** - барьер и возвращает !0 только если для всех потоков блока predicat != 0;
- **`int __syncthreads_or(int predicate)`** - барьер и возвращает !0 если хотя бы для одного потока блока predicat != 0.

Синхронизация памяти

При работе с современными вычислительными системами с общей памятью необходимо учитывать тот факт, что скорость работы подсистемы памяти меньше скорости работы процессора, а также наличие кэш-памяти. В связи с этим, во многих моделях параллельных вычислений принимается принцип ослабленной согласованности памяти. Данный принцип утверждает, что потоки временно могут «видеть» предыдущие значения переменных других потоков (до фактической записи переменных).

В CUDA имеются следующие функции явной синхронизации памяти:

- **`void __threadfence_block()`** - ожидание до тех пор пока результаты предшествующих операций с глобальной и разделяемой памятью, совершаемых вызывающим потоком, не будут видны всем потокам в пределах блока;
- **`void __threadfence()`** - ожидание до тех пор пока результаты предшествующих операций с глобальной и разделяемой памятью, совершаемых вызывающим потоком, не будут видны следующим потокам:
  - для операций с разделяемой памятью - потокам в пределах блока;
  - для операций с глобальной памятью - потокам в пределах ГПУ.
- Синхронизация памяти (для архитектуры 2.x):
- **`void __threadfence_system()`** - ожидание до тех пор пока результаты предшествующих операций с глобальной и разделяемой

памятью, совершаемых вызывающим потоком, не будут видны следующим потокам:

- для операций с разделяемой памятью - потокам в пределах блока;
- для операций с глобальной памятью - потокам в пределах ГПУ;
- для операций с page-locked памятью ЦПУ - потокам ЦПУ.

Атомарные инструкции

Атомарные инструкции необходимы для предотвращения гонок данных и реализуются в CUDA через функции.

Обычная последовательность действий с памятью: прочитать содержимое памяти, изменить его и записать обратно.

Гарантируется, что при использовании атомарных инструкций потоки не будут взаимодействовать между собой. Например, после прочтения переменной 1-м потоком, 2-й поток не получит к ней доступ до записи результата 1-м потоком.

Атомарные инструкции можно применять для доступа к разделяемой и глобальной памяти.

Допустимые варианты атомарных инструкций достаточно сильно различаются для ГПУ разной архитектуры.

В состав атомарных инструкций входят:

- арифметические операции: atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS;
- побитовые операции: atomicAnd, atomicOr, atomicXor.

Пример использования синхронизации: редукция массива суммированием

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float *array, unsigned int N, float
*result)
{
     float partialSum = calculatePartialSum(array, N);
     if (threadIdx.x == 0) {
          result[blockIdx.x] = partialSum;
          __threadfence();
          unsigned int value = atomicInc(&count, gridDim.x);
          isLastBlockDone = (value == (gridDim.x - 1));
     }
     __syncthreads();
     if (isLastBlockDone) {
          float totalSum = calculateTotalSum(result);
          if (threadIdx.x == 0) {
               result[0] = totalSum;
               count = 0;
```

```
        }
    }
}
```

Глобальная синхронизация

Глобальная синхронизация – это синхронизация всех потоков в пределах ГПУ. Данный вид синхронизации часто требуется при решении задач математического моделирования, рассматривающих эволюцию физической системы во времени, также в итерационных алгоритмах линейной алгебры и пр. В этих случая требуется синхронизация между очередными итерациями алгоритма. Для этого могут быть использованы атомарные инструкции (функции), что продемонстрировано выше.

Также задача глобальной синхронизации может решаться при помощи взаимодействия ЦПУ и ГПУ с использованием барьерной функции `cudaDeviceSynchronize()`:

```
for (t = 1; t <= Nt; t++)
{
  HeatTransfer_Step <<<dimGrid, dimBlock>>> (T0d, Td, Nx0,
Ny0);
  cudaDeviceSynchronize(); // синхронизация между итерациями
  swap(T0d, Td);
}
```

Стоит отметить, что вызов ядра (kernel) асинхронен! Т.е. после запуска ядра ЦПУ продолжает выполнение собственной программы, не дожидаясь завершения работы ГПУ. Для корректной работы данного фрагмента требуется явная барьерная синхронизация между итерациями.

Заключение: изучение модельной задачи

Рассмотрим двумерное уравнение теплопроводности. Будем решать уравнение при помощи явной разностной схема с 5-точечным шаблоном типа «крест»:

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \alpha\Delta t\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right);$$

$$i = \overline{1,m_x};\ j = \overline{1,m_y};\ n = \overline{0,m_t};$$

Рассмотрим несколько подходов к реализации вычислений на ГПУ.

Способ 1

Каждый узел сетки обрабатывает один поток, копируя необходимые для этого данные из медленной оперативной памяти независимо от других потоков. Недостаток этого подхода очевиден: отсутствует повторное использование уже принятых данных. Почти все внутренние узлы сетки будут скопированы из памяти 4 раза (вследствие использования шаблона «крест»).

Количество читаемых из оперативной памяти элементов массива:

$5(n_x-4)(n_y-4) + 2\cdot4(n_x-4) + 2\cdot4(n_y-4) + 3\cdot4 + 2\cdot(n_x-2) + 2\cdot(n_y-2) \approx 5n_xn_y$, при больших n_x и n_y.

Количество записываемых в оперативную память элементов массива: n_xn_y. Таким образом, количество пересылок приблизительно равно $6n_xn_y$.

В идеальном случае (при однократном копировании исходного массива) количество пересылок составит $2n_xn_y$. Поскольку копирование данных занимает время на 2 порядка большее, чем выполнение операции умножить-сложить (MAD), то максимальный ресурс ускорения вычислений составляет около $6n_xn_y/(2n_xn_y) = 3$ раза.

Способы 2 и 3

Доработаем первый способ следующим образом. Организуем потоки блока в двумерную структуру (для индексации потока внутри блока используется два целых числа). Блоки потоков организуем в двумерную сетку, так что каждому узлу расчётной области по-прежнему соответствует один поток. При этом каждому блоку потоков будет соответствовать своя двумерная подобласть.

Каждый блок потоков предварительно копирует вычисляемую двумерную подобласть в быструю разделяемую (shared) память, выполняет синхронизацию, гарантирующую окончание процесса копирования, затем каждый поток внутри блока вычисляет свой узел сетки и сохраняет рассчитанное значение в медленную память. Таким образом, данный способ предусматривает ручную имитацию кэш-памяти, позволяющей повторно использовать полученные из медленной памяти данные.

Для расчёта каждой подобласти необходимо использовать граничные значения соседних подобластей. Здесь возникает два варианта реализации.

Способ 2. Все потоки блока копируют из глобальной памяти подобласть с граничными узлами, а затем потоки, соответствующие внутренним ячейкам подобласти выполняют расчёт. При использовании размера подобласти 32x16, конфигурация блока потоков следующая - 34x18. Дальнейшее увеличение размеров подобласти наталкивается на аппаратное ограничение количества потоков в блоке.

Способ 3. Все потоки блока копируют из глобальной памяти в разделяемую память внутренние узлы подобласти, затем часть потоков копирует граничные узлы из глобальной памяти. При использовании

размера подобласти 32x16, конфигурация блока потоков следующая — 32x16.

В случае способа 2 запросы на копирование из глобальной памяти получаются несгруппированными (uncoalesced) в силу того, что в пределах некоторых полуварпов (halfwarp) адреса копируемых элементов не обладают свойством близости.

У способа 3 также есть недостаток: наличие программных ветвлений для потоков в пределах варпа, замедляющих работу вычислителя (ГПУ приходится выполнять каждую ветвь ветвления, поскольку все потоки варпа выполняют одну и ту же инструкцию).

Способ 4

Поскольку блок потоков, в силу ограничений архитектуры, не может скопировать данные из своей разделяемой памяти в разделяемую память другого блока потоков, то неизбежно повторное копирование данных из глобальной памяти на границах подобластей. В случае способов 2 и 3 повторному копированию подвергаются столбцы и строки на границах подобластей. В силу аппаратных ограничений, для сеток, имеющих порядка 10^8 узлов, количество активных блоков потоков для предлагаемой выше схемы распараллеливания много меньше общего количества блоков в сетке. Поэтому, изменив схему декомпозиции расчётной области можно сократить количество повторно копируемых элементов, не ухудшив параллелизма. Особенно актуально исключение повторного копирования элементов столбцов двумерного массива (в используемом языке программирования Си элементы матриц хранятся в памяти по строкам), поскольку уменьшается доля несгрупированных запросов к глобальной памяти ГПУ (относительно способа 3, который взят за основу для способа 4; поскольку способ 3 быстрее - см. ниже в п. 4).

Проведём одномерную декомпозицию расчётной области. При этом каждому блоку потоков достаётся полоса расчётной области, состоящая из нескольких строк массива, т.е. формируется одномерная сетка блоков. Блок потоков остаётся двумерным, его работа сводится к последовательному расчёту прямоугольных частей (размерностью блока) доставшейся ему подобласти. Наглядно это можно представить как перемещение рассчитываемой «плитки» вдоль столбцов выделенной полосы массива. При этом устраняется повторное копирование столбцов массива за счёт того, что на каждой итерации внутри подобласти (перемещение «плитки» на один шаг) граничные узлы передаются из последнего столбца массива, расположенного в разделяемой памяти, в первый столбец.

При тестировании программной реализации алгоритмов использовалась ЭВМ следующей конфигурации: четырёхядерный ЦПУ Intel Core2 Quad 9550 2,83 ГГц с поддержкой HyperThreading, 12 Гбайт ОЗУ, ГПУ NVidia GTX 260 (24 восьмиядерных мультипроцессоров, общее количество потоковых ядер 192; 1,3 ГГц; 2 Гб ОЗУ).

Размерность области 14400x14400, расчёты с одинарной точностью (двойная точность на данной версии ГПУ поддерживается недостаточно), количество шагов по времени — 100 (синхронизация между итерациями посредством ЦПУ (функция cudaDeviceSynchronize)), размерность блока потоков 32x16.

Результаты расчётов приведены в таблице 1, ускорение вычислено относительно способа 1. Время передачи данных между ЦПУ и ГПУ — 0,6 с.

Таблица 1

	Способ 1	Способ 2	Способ 3	Способ 4
Время вычислений на ГПУ (без учёта обмена данными ЦПУ-ГПУ), с	7,41	6,26	5,96	4,03
Ускорение	1	1,17	1,23	1,85