Массивный параллелизм и синхронизация в CUDA на примере решения системы ОДУ большой размерности

Дмитрий Глызин
http://parallel.ncycle.org/

CUDA: массивный параллелизм

- Большое количество вычислительных ядер (192 на чипсете GTX260)
- Огромное количество одновременно выполняемых тредов (более 30 млн)
- Переключение между тредами не затратно, треды «легкие»

GPU: физическая структура

- Ядра объединены группами по 8 в потоковых мультипроцессорах
- Каждый мультипроцессор одновременно обрабатывает 32 треда (т.н. warp), разбивая их на группы по 8
- Ветвления внутри warp-а = замедление

CUDA: логическая структура

- На логическом уровне треды объединены в блоки
- Внутри блока нити легко синхронизируются
- Максимальное число нитей в блоке - 512*
- Максимальное число блоков в сети - 65535*
- Уложить в один блок задачу, требующую синхронизации тредов, удается не всегда

*на примере чипсета GTX260

CUDA: память

- В DRAM расположены:
  - Глобальная, доступна и из host-а, и из device-а
  - Локальная, доступна только из треда
  - Текстурная и константная (кэшируемые)
- На самом мультипроцессоре расположены:
  - Общая, доступна для каждого треда из блока
  - Регистровая, локальная для каждого треда

CUDA: синхронизация

- Внутри блока синхронизация не затратна: syncthreads()
- Синхронизация между блоками происходит только при завершении kernel-а, то есть требует его перезапуска
- Необходимо минимизировать синхронизации между блоками

Решение больших систем ОДУ

- Применение методов класса Рунге-Кутты в параллельных вычислениях в общем случае ограничено (требует за шаг многократной глобальной синхронизации)
- Тем не менее, для задач, полученных из PDE методом прямых, структура правой части допускает эффективную параллелизацию с помощью CUDA

Решение больших систем ОДУ

В качестве модели рассмотрим решение методом Рунге-Кутты четвертого порядка следующей задачи:

$$\dot{u}_i = u_i + d(u_{i+1} + u_{i-1} - 2u_i + a(v_{i+1} + v_{i-1} - 2v_i)) - (u_i^2 + v_i^2)(u_i - bv_i),$$

$$\dot{v}_i = v_i + d(-a(u_{i+1} + u_{i-1} - 2u_i) + v_{i+1} + v_{i-1} - 2v_i) - (u_i^2 + v_i^2)(bu_i + v_i),$$

$$1 \le i \le n-2,$$

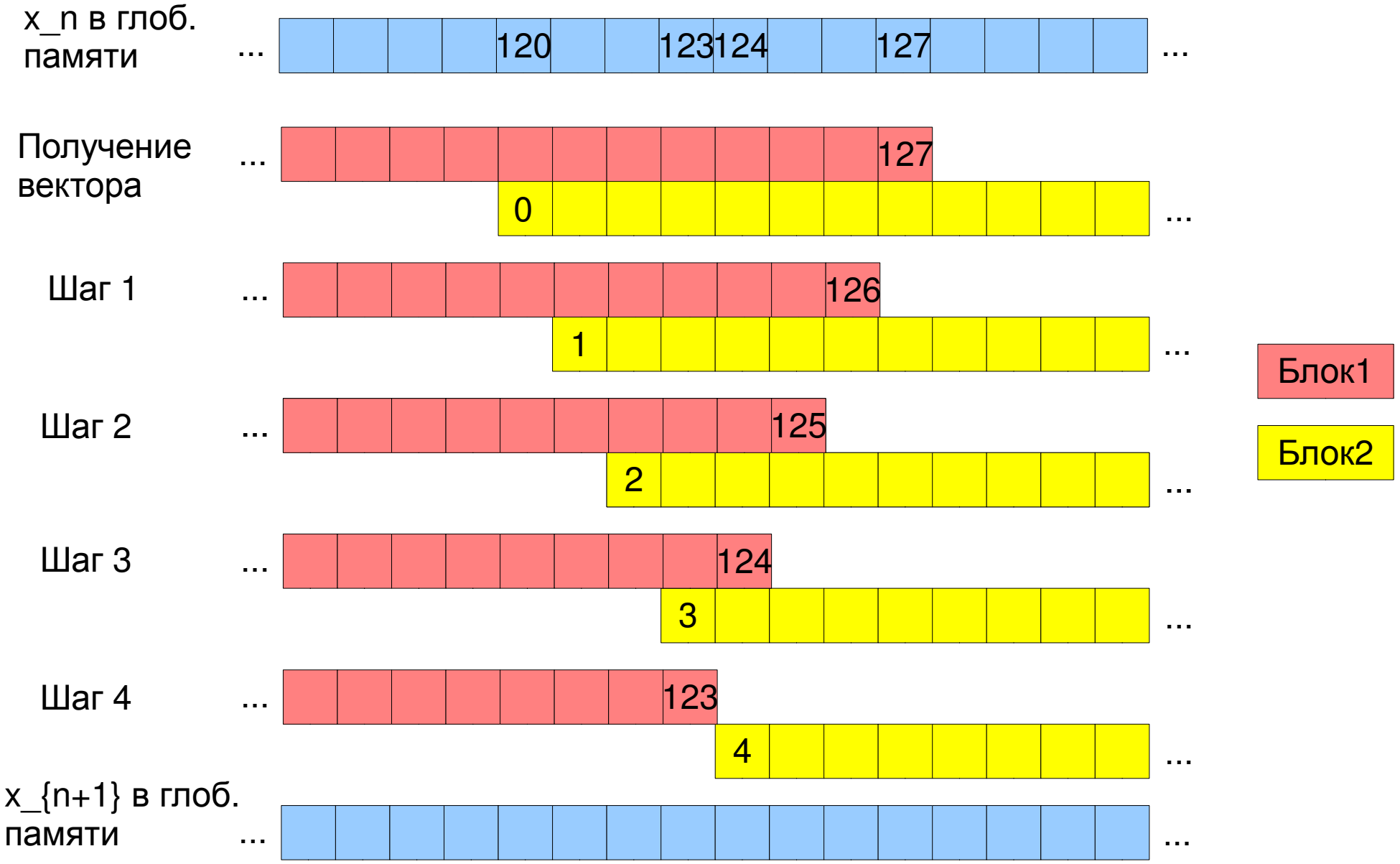
(при $i = 0$ и $i = n - 1$ учитываются краевые условия), полученной методом прямых из одномерного комплексного уравнения Гинзбурга-Ландау с краевыми условиями Неймана:

$$\frac{\partial W}{\partial t} = W + d(1 + ia)\frac{\partial^2 W}{\partial x^2} - (1 + ib)|W|^2 W$$

Реализация метода

- Каждый тред отвечает за два элемента
- Вычисление i-й правой части требует знания лишь соседних элементов вектора
- Для крайних в блоке тредов эффективнее повторно вычислить значение, чем узнать готовый результат у треда из соседнего блока
- За каждый из четырех этапов шага метода блок теряет информацию об одной компоненте вектора с каждой стороны
- Если мы организуем перекрытие в 8 элементов рассылки начального вектора, то на выходе шага получим вектор целиком, не производя синхронизацию между блоками

Реализация метода



Выдержки из кода kernel-a

```
//----правая часть решаемого уравнения----//
__device__ void ginzland ( double * s1, double* s2, double * r1, double* r2, int idx)
{                                          //нулевой элемент локального массива — это элемент массива в
                                           //shared памяти с индексом, равным номеру нити,
                                           //поэтому возможно обращение к отрицательным номерам
  double norm;
  norm = s1[0]*s1[0] + s2[0]*s2[0];

  if (idx == 0)   //Неизбежные ветвления, связанные с краевыми условиями
  {
    *r1 = D * (s1[1]-s1[0] + A*(s2[1]-s2[0])) + s1[0] - norm * (s1[0]-B*s2[0]);
    *r2 = D * (-A*(s1[1]-s1[0]) + s2[1]-s2[0]) + s2[0] - norm * (B*s1[0]+s2[0]);
  }

  if (idx == SYSDIM-1)
  {
    *r1 = D*(s1[-1]-s1[0] + A*(s2[-1]-s2[0])) + s1[0] - norm * (s1[0]-B*s2[0]);
    *r2 = D*(-A*(s1[-1]-s1[0]) + s2[-1]-s2[0])+s2[0]-norm * (B*s1[0]+s2[0]);
  }

  if ((idx>0)&&(idx<SYSDIM-1))
  {
    *r1 = D * (s1[1]+s1[-1]-2*s1[0] + A*(s2[1]+s2[-1]-2*s2[0])) + s1[0] - norm * (s1[0]-B*s2[0]);
    *r2 = D * (-A*(s1[1]+s1[-1]-2*s1[0]) + s2[1]+s2[-1]-2*s2[0]) + s2[0] - norm * (B*s1[0]+s2[0]);
  }
}
```

Выдержки из кода kernel-а

```
//Шаг метода Рунге-Кутты
__global__ void RK4_STEP  ( double *y_glo, double *res_glo )
{
  int idx = blockIdx.x*(blockDim.x-8)+threadIdx.x;
  int idx2 = idx+SYSDIM;
  int lbound = threadIdx.x;
  int rbound = threadIdx.x;
  __shared__ double y[BLOCKDIM];
...//здесь объявляются все общие массивы
  __shared__ double r3[BLOCKDIM];

if (idx<SYSDIM)
{
  y[threadIdx.x] = y_glo[idx];
  y2[threadIdx.x] = y_glo[idx2];
  __syncthreads();
//первая стадия шага
  ginzland(&y[threadIdx.x], &y2[threadIdx.x], &k1[threadIdx.x], &r1[threadIdx.x], idx);
  k1[threadIdx.x] = y[threadIdx.x]+y[threadIdx.x+1]+y[threadIdx.x-1];
  r1[threadIdx.x] = y2[threadIdx.x]+y2[threadIdx.x+1]+y2[threadIdx.x-1];
  s1[threadIdx.x] = y[threadIdx.x] + STEP*k1[threadIdx.x]/2.0;
  s2[threadIdx.x] = y2[threadIdx.x] + STEP*r1[threadIdx.x]/2.0;
  __syncthreads();
...//здесь происходят еще три стадии шага
  __syncthreads();
  res_glo[idx] = k1[threadIdx.x];
  res_glo[idx2] = r1[threadIdx.x];
}
}
```

Обращение к памяти

В программе доступ к памяти организован с учетом следующих особенностей архитектуры CUDA:

- Coalescing: все чтения из глобальной памяти в каждых 16-ти подряд идущих тредах обращаются к одному сегменту в 128 байтов, и каждый тред читает 8-байтовое слово
- Bank conflict общей памяти минимизирован до второго уровня в случае double- и отсутствует в случае single-арифметики

Итоги

- Даже при условии, что задача не идеально подходит для массивно-параллельного выполнения, ее реализация с помощью CUDA дает значительный эффект
- В данном случае несущественное увеличение общего объема вычислений позволило снизить вчетверо издержки на глобальную синхронизацию, и получить результат, превосходящий современный высокопроизводительный CPU

Итоги

- Алгоритм решения приведенной задачи был реализован с помощью CUDA на карте GTX260 и с помощью OpenMP на четырехядерном CPU Intel Core i7-920: ускорение на видеокарте составило 2.31 раз для double- и 8.30 раз для single-арифметики
- Для теста производительности выполнялось 30000 шагов метода для системы размерности 32768, по 128 нитей (256 уравнений) в блоке.