Центр микро- и наномасштабной динамики дисперсных систем

Модель памяти GPU/CUDA

Марьин Д. Ф.

Уфа, 2011г.

Синхронизация

Для синхронизации текущей нити на CPU с GPU используется функция:

```
cudaError_t  cudaThreadSynchronize ( void );
```

CUDA поддерживает синхронизацию потоков в одном блоке на GPU:

```
__sincthreads ();
```

Примечание:

- каждый поток задает последовательность операций, выполняемых в строго определенном порядке;
- порядок выполнения операций между разными потоками не является строго определенным.

Для отслеживания выполнения кода на GPU в CUDA используются event'ы.

```
cudaEvent_t startEvent;
```

Создание и уничтожение event'а

```
cudaError_t cudaEventCreate  ( cudaEvent_t * event );
cudaError_t cudaEventDestroy ( cudaEvent_t   event );
```

Задание места, прохождение которого должен сигнализировать данный event.

```
cudaError_t cudaEventRecord ( cudaEvent_t event,
                              CUstream stream );
```

Если stream не равен 0, то отслеживается только завершение выполнения всех операций в данном потоке.
Этот запрос является асинхронным — он только обозначает место в потоке команд, прохождение которого потом будет запрашиваться.

```
cudaError_t cudaEventQuery ( cudaEvent_t event );
```

выполняет мгновенную проверку "прохождения"данного event'а — управление из нее сразу же возвращается.
В случае, если все операции, предшествующие данному event'у были закончены — возвращается cudaSuccess, иначе — cudaErrorNotReady.

```
cudaError_t cudaEventSynchronize ( cudaEvent_t event );
```

Обеспечивает явную синхронизацию (блокировку) — ожидание пока все операции для данного event'а не будут завершены.

```
cudaError_t cudaEventElapsedTime ( float * time,
      cudaEvent_t startEvent, cudaEvent_t stopEvent );
```

позволяет узнать время в мс (с точностью до $\frac{1}{2}$ мкс), прошедшее между данными event'ами (между моментами, когда каждый из этих event'ов был "записан").

Пример замера времени

```
cudaEvent_t start, stop;

cudaEventCreate ( &start );
cudaEventCreate ( &stop );

cudaEventRecord ( start, 0 );
...
// execution of kernel
...
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &gpuTime, start, stop );

printf ( "Elapsed time: %.2f ms\n", gpuTime );
```

Shared memory — 16КБ блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре.

- Расположение: **multiprocessor**
- Кэшируемость: **no**
- Уровень выделения: **on chip**
- Доступ: **GPU — R/W, CPU — no**
- Скорость работы: **высокая**
- Уровень доступа: **per-block, all SP in MP**
- Время жизни: **block**

— Адресуется одинаково для всех потоков внутри блока.
— Управляется разработчиком напрямую. Используется в качестве управляемого программистом кэша первого уровня.
— От того, сколько разделяемой памяти требуется блоку, зависит количество блоков, которое может быть запущено на одном мультипроцессоре.

Примечание

Разделяемая память используется для передачи параметров при запуске ядра на выполнение, поэтому следует избегать передачи большого объема данных, непосредственно передаваемых ядру в конструкции вызова ядра.

1ый способ: явное задание размеров массива

Компилятор сам произведет выделение необходимого количества разделяемой памяти для каждого блока при запуске ядра

```
__global__ void kernel (float * a)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;

//256 * sizeof(float) bytes per block
__shared__ float shared_buf [256];

// copying: shared <- global
shared_buf [threadIdx.x] = a [index];

__syncthreads();
...
}
```

2ой способ: без явного задания размеров массива

При вызове ядра задается дополнительный объём разделяемой памяти (в байтах), который необходимо выделить каждому блоку при запуске ядра.

```
kernel <<< dim3(n/256), dim3(256),
           k*sizeof(float) >>> (a);
```

При запуске ядра начало такого массива будет совпадать с началом дополнительно выделенной памяти.

2ой способ: без явного задания размеров массива

Для доступа к такой памяти используется описание массива без явного задание размера.

```
__global__ void kernel (float * a)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;

__shared__ float shared_buf [];

// copying: shared <- global
shared_buf [threadIdx.x] = a [index];

__syncthreads();
...
}
```

2ой способ. Примечание

Можно задать несколько массивов в разделяемой памяти без явного указания их размеров, но тогда в момент выполнения ядра они будут расположены в начале выделенной блоку разделяемой памяти, т.е. их начала совпадут.
В этом случае на программиста ложится ответственность за явное разделение памяти между такими массивами.

```
__global__ void kernel ( float * a, int k )
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
__shared__ float shared_buf1 [];
__shared__ float shared_buf2 [];
shared_buf1 [threadIdx.x]     = a [index];
shared_buf2 [threadIdx.x + k] = a [index + k];
__syncthreads();
...
}
```

Задача об N телах

```
#define  N                  (16*1024)
#define  BLOCK_SIZE          256

int main ( int argc, char * argv [] )
{
float3  * pos      = new float4 [N];
float3  * vel      = new float4 [N];
float3  * pDev [2] = { NULL, NULL };
float3  * vDev [2] = { NULL, NULL };

cudaEvent_t start, stop;
cudaEventCreate ( &start );
cudaEventCreate ( &stop );
cudaEventRecord ( start, 0 );
```

Задача об N телах

```
// read or generate initial pos and vel
...

cudaMalloc ( (void **) &pDev[0], N * sizeof (float4) );
cudaMalloc ( (void **) &vDev[0], N * sizeof (float4) );
cudaMalloc ( (void **) &pDev[1], N * sizeof (float4) );
cudaMalloc ( (void **) &vDev[1], N * sizeof (float4) );

cudaMemcpy ( pDev[0], pos, N * sizeof (float4),
             cudaMemcpyHostToDevice );
cudaMemcpy ( vDev[0], vel, N * sizeof (float4),
             cudaMemcpyHostToDevice );
```

Задача об N телах

```
int     index      = 0;
float   dt         = 0.01f;
int     nTimeSteps = 100;
for ( int i = 0; i < nTimeSteps; i++, index ^= 1 )
    integrateBodies <<< dim3(N/BLOCK_SIZE),
                        dim3(BLOCK_SIZE) >>>
        ( pDev [index^1], vDev [index^1],
          pDev [index], vDev [index], dt );

cudaMemcpy ( pos, pDev [index^1], N * sizeof ( float4 ),
             cudaMemcpyDeviceToHost );
cudaMemcpy ( vel, vDev [index^1], N * sizeof ( float4 ),
             cudaMemcpyDeviceToHost );

cudaFree   ( pDev [0] );
cudaFree   ( vDev [0] );
cudaFree   ( pDev [1] );
cudaFree   ( vDev [1] );
```

Задача об N телах

```
float     gpuTime = 0.0f;

cudaEventRecord       ( stop, 0 );
cudaEventSynchronize  ( stop );
cudaEventElapsedTime  ( &gpuTime, start, stop );

printf ( "GPU Elapsed time: %.2f ms\n", gpuTime );

...

delete pos;
delete vel;

return 0;
}
```

Задача об N телах. Ядро

```
__global__ void integrateBodies (
           float4 * newPos, float4 * newVel,
           float4 * oldPos, float4 * oldVel, float dt )
{
 int index  = blockIdx.x * blockDim.x + threadIdx.x;
 float3 pos = oldPos [index];
 float3 f   = make_float3 ( 0.0f, 0.0f, 0.0f );

 for ( int i = 0; i < N; i++ ) {
  if (i==index) continue;
  float3 pi = oldPos [i];
  float3 r;
  r.x = pi.x-pos.x; r.y = pi.y-pos.y; r.z = pi.z-pos.z;
  float invDist = 1.0f /
      sqrtf (r.x*r.x + r.y*r.y + r.z*r.z + EPS*EPS);
  float s =  invDist * invDist * invDist;
  f.x += r.x*s; f.y += r.y*s; f.z += r.z*s;
 }
```

Задача об N телах. Ядро

```
float3 vel = oldVel[index];
vel.x += f.x   * dt;
vel.y += f.y   * dt;
vel.z += f.z   * dt;

pos.x += vel.x * dt;
pos.y += vel.y * dt;
pos.z += vel.z * dt;

newPos[index] = pos;
newVel[index] = vel;
}
```

Задача об N телах. Оптимизация ядра

Для каждого тела необходимо сделать N-1 чтений положений других тел.
Стандартный приём: каждому блоку выделяется массив в разделяемой памяти размером, равным размеру блока.

```
__global__ void integrateBodies (
           float4 * newPos, float4 * newVel,
           float4 * oldPos, float4 * oldVel, float dt )
{
 int   index = blockIdx.x * blockDim.x + threadIdx.x;
 float3 pos = oldPos [index];
 float3 f   = make_float3 ( 0.0f, 0.0f, 0.0f );
 int    ind = 0;
 __shared__ float4 sp [BLOCK_SIZE];
```

Задача об N телах. Оптимизация ядра

```
for ( int i=0; i<N/BLOCK_SIZE; i++, ind += BLOCK_SIZE )
{
 sp [threadIdx.x] = oldPos [ind + threadIdx.x];
 __syncthreads ();

 for ( int j = 0; j < BLOCK_SIZE; j++ ) {
  if (j==index) continue;
  float3 r;
  r.x = sp[j].x-pos.x;
  r.y = sp[j].y-pos.y;
  r.z = sp[j].z-pos.z;
  float invDist = 1.0f /
       sqrtf (r.x*r.x + r.y*r.y + r.z*r.z + EPS*EPS);
  float s = invDist * invDist * invDist;
  r.x += r.x*s;  f.y += r.y*s;  f.z += r.z*s;
  }
  __syncthreads ();
}
```

Задача об N телах. Оптимизация ядра

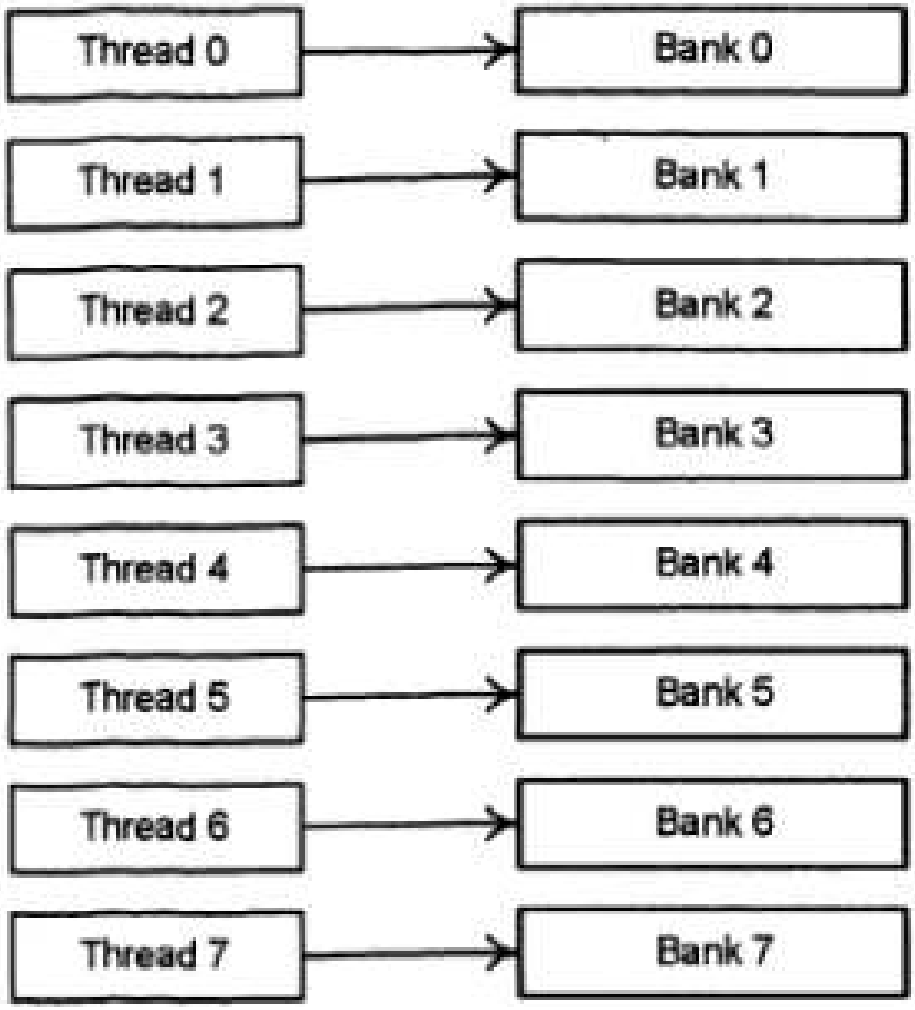
```
    float3 vel = oldVel[index];

    vel.x += f.x   * dt;
    vel.y += f.y   * dt;
    vel.z += f.z   * dt;

    pos.x += vel.x * dt;
    pos.y += vel.y * dt;
    pos.z += vel.z * dt;

    newPos[index] = pos;
    newVel[index] = vel;
}
```

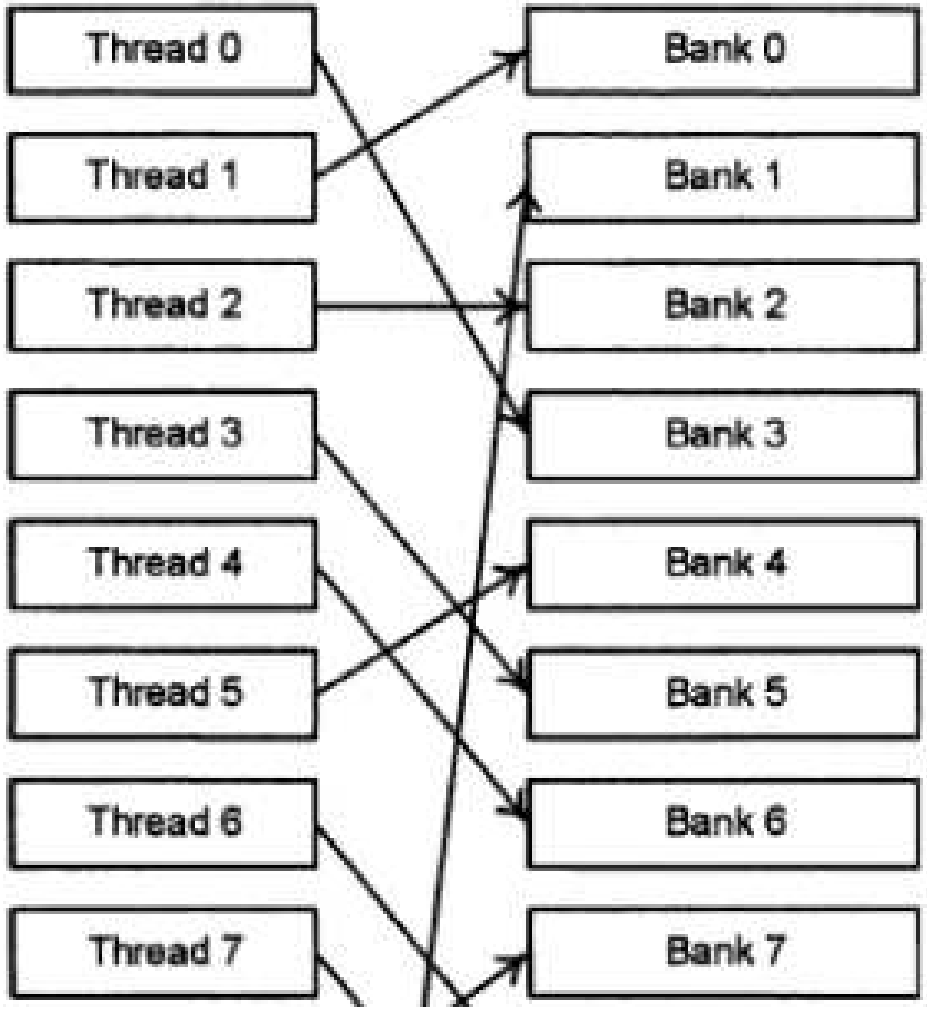
Паттерны доступа к разделяемой памяти

Для повышения пропускной способности вся разделяемая память разбита на 16 банков, каждый из которых способен выполнить одно чтение/запись 32-битового слова.
Бесконфликтный доступ:

- Все 16 нитей полуварпа обращаются к 16 32-битовым словам, лежащим в разных банках (порядок не важен). Если в один банк придёт сразу несколько обращений, то он выполнит их последовательно — конфликт банков.
- Все 16 нитей полуварпа обращаются к одному и тому же адресу — broadcasting.

Бесконфликтный доступ к разделяемой памяти





Доступ к разделяемой памяти с возникновением конфликтов