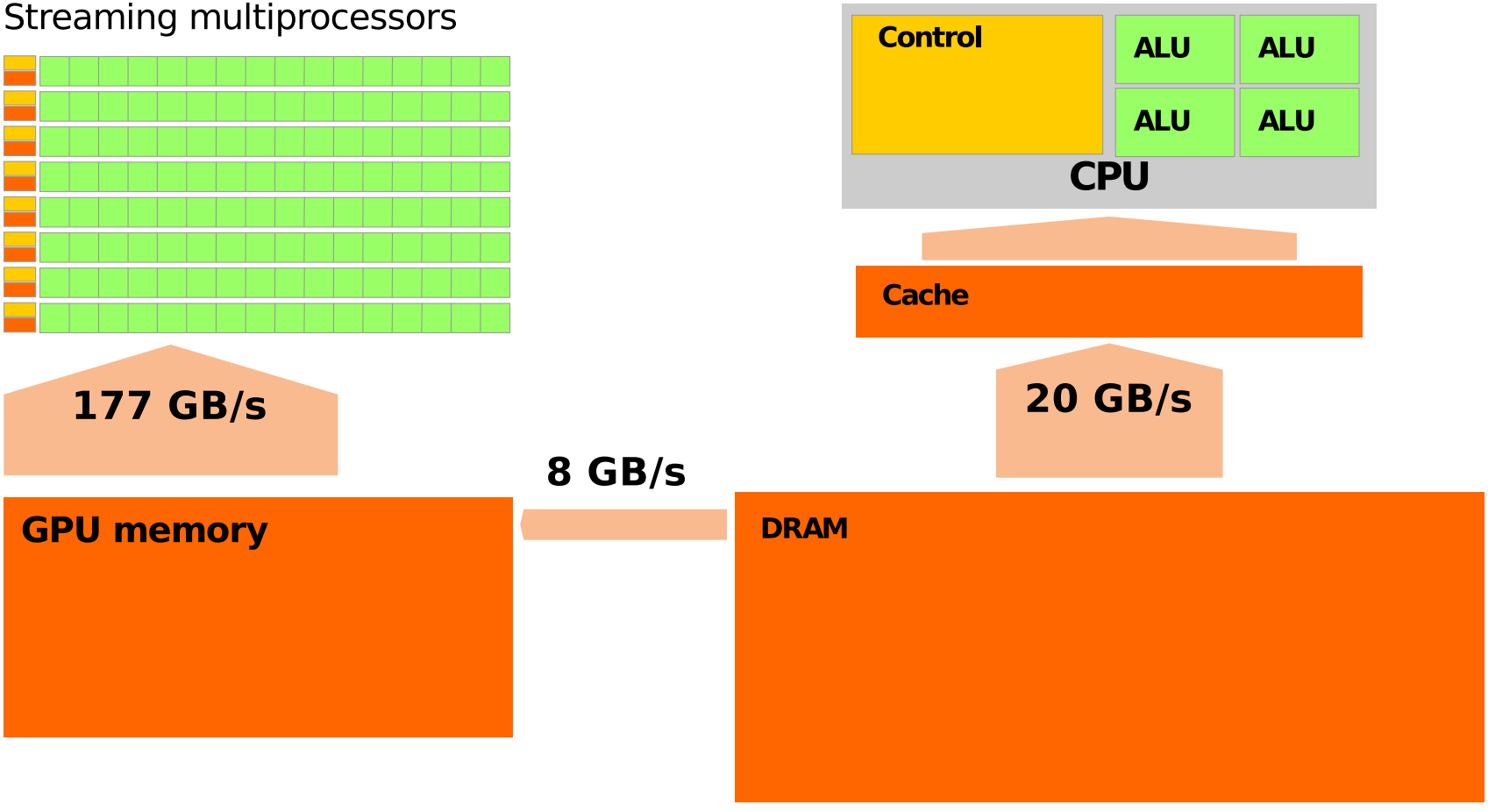
Пример программы, вычисляющей решеточное действие с помощью CUDA

А. Худяков

16 октября 2012 г.

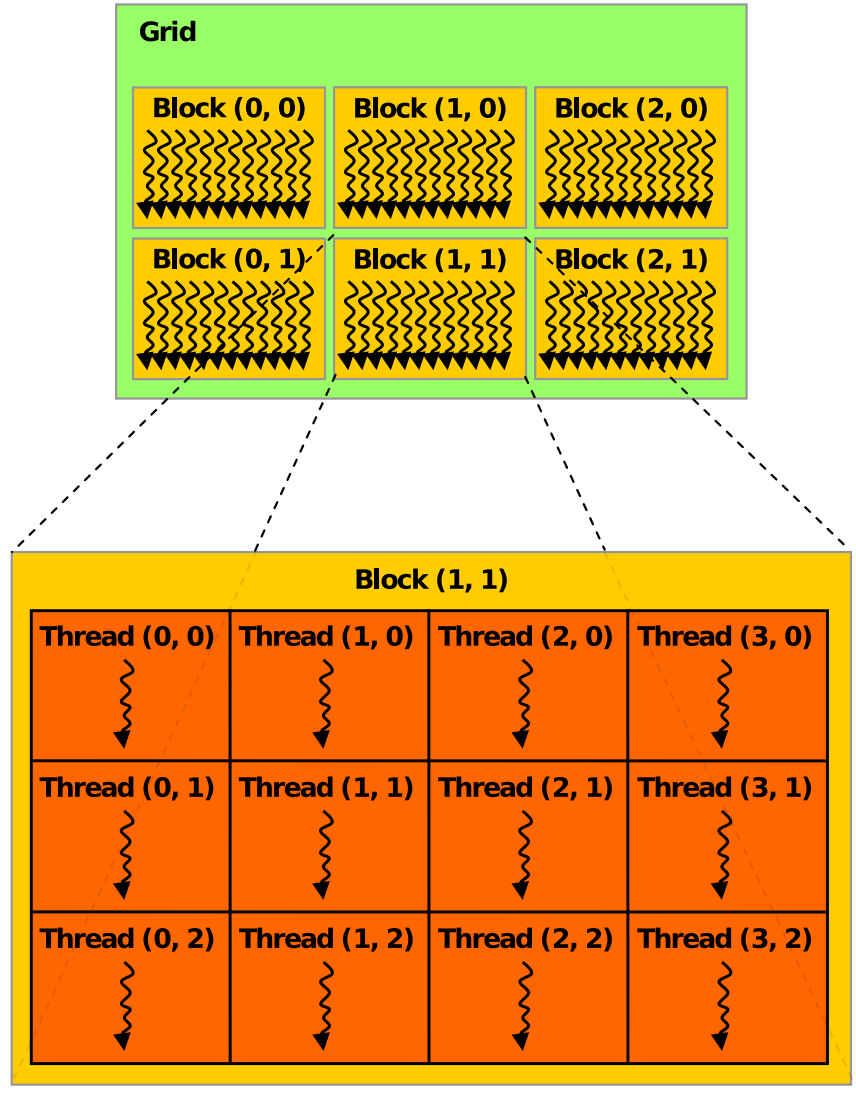
Архитектура GPU



Threads blocks and grids

Треды на GPU собраны в иерархическую структуру:

- Треды группируются в блоки (blocks), те в свою очередь, группируются в решётку (grid).
- Каждый тред в блоке выполняет одну и ту же функцию.



Индексирование тредов и блоков

Каждый тред в блоке и каждый блок имеют индекс. Также для каждого блока доступен его размер. Их можно получить из специальных переменных:

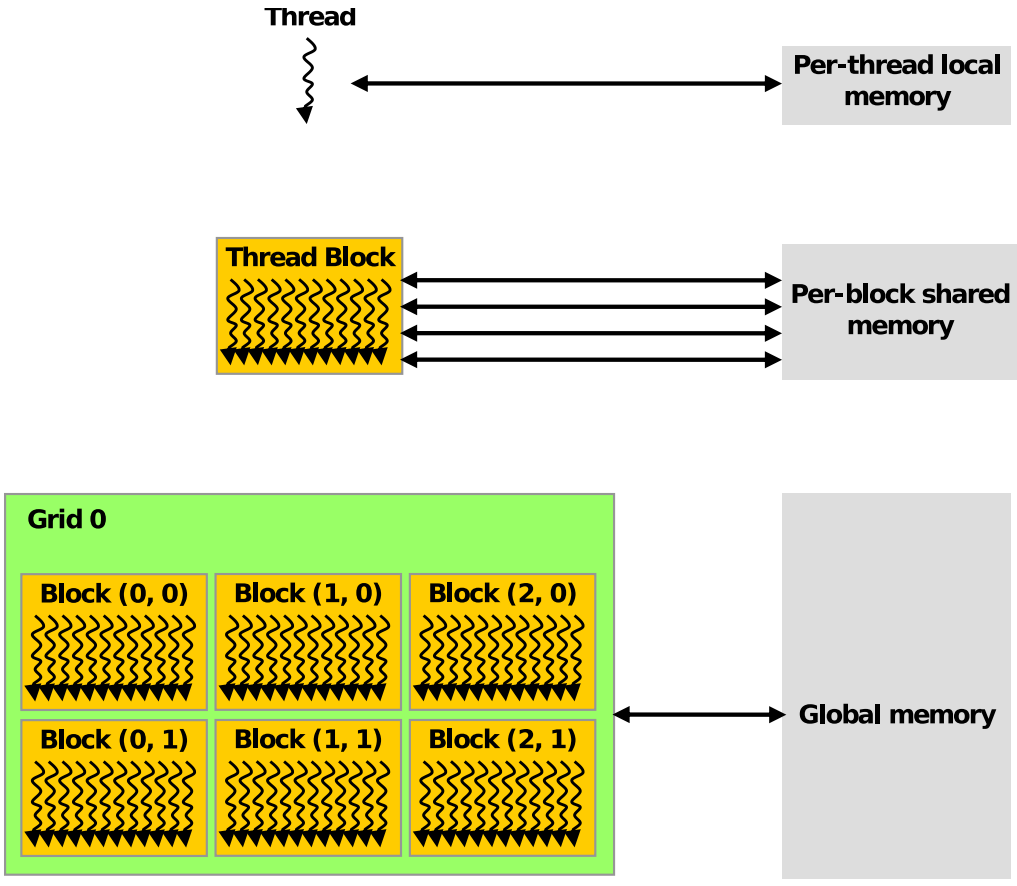
- `threadIdx` — индекс треда
- `blockIdx` — индекс блока
- `blockDim` — размер блока

Для удобства работы с 2D и 3D данными эти переменные являются трехкомпонентным массивом:

```
threadIdx.x, threadIdx.y, threadIdx.z
```

В одномерном случае используется только одна компонента `x`

Иерархия памяти



Иерархия памяти

	Location on/off chip	Cached	Scope	Lifetime
Register	On	n/a	1 thread	thread
Local	Off	yes*	1 thread	thread
Shared	On	n/a	block	block
Global	Off	yes*	all thread + host	Host allocation
Constant	Off	yes	all thread + host	Host allocation
Texture	Off	yes	all thread + host	Host allocation

* — кешируется только устройтсвами с computing capability >2.x

Локальная память импользуется только для auto переменных, если они не помещаются в регистры. Она медленней чем shared!

Аллокация shared memory

Статически известный размер

```
__global__ kernel( ... ) {
  __shared__ double sdata[256];
}
```

Динамическая алокация

```
__global__ kernel( ... ) {
  extern __shared__ double sdata[];
}
```

```
kernel<<<gridSize,nBlocks,sh_size>>>kernel( ... )
```

`sh_size` — размер shared memory в байтах

Concurrency & shared memory

Concurrency

- Треды в разных блоках не могут общаться.
- Треды в одном блоке могут передавать информация через shread memory.

Race conditions

Если два треда пишут в одну и ту же область памяти — результат неопределён!

Необходима синхронизация тредов!

Синхронизация

Для синхронизации используется барьеры:

```
__syncthreads()
```

Гарантирует, что все треды в блоке дойдут до этой точки прежде чем продолжить исполнение.

Пример

```
sdata[tid] = a[bid * bsize + tid];
__syncthreads();
// Do something
```

Вычисление действия на решётке

Действие на 1+1D:

$$S = \int dtdx \left(\frac{(\partial_\tau \phi)^2}{2} + \frac{(\partial_x \phi)^2}{2} + V(\phi) \right)$$

$$V(\phi) = \frac{\lambda}{2}(\phi^2 - 1)^2 - \varepsilon\phi$$

$$\phi_B(\tau, x) = \mathrm{th}(\sqrt{\lambda}(r - R_0))$$

Будем считать на решётке 2048×2048.

Общая структура

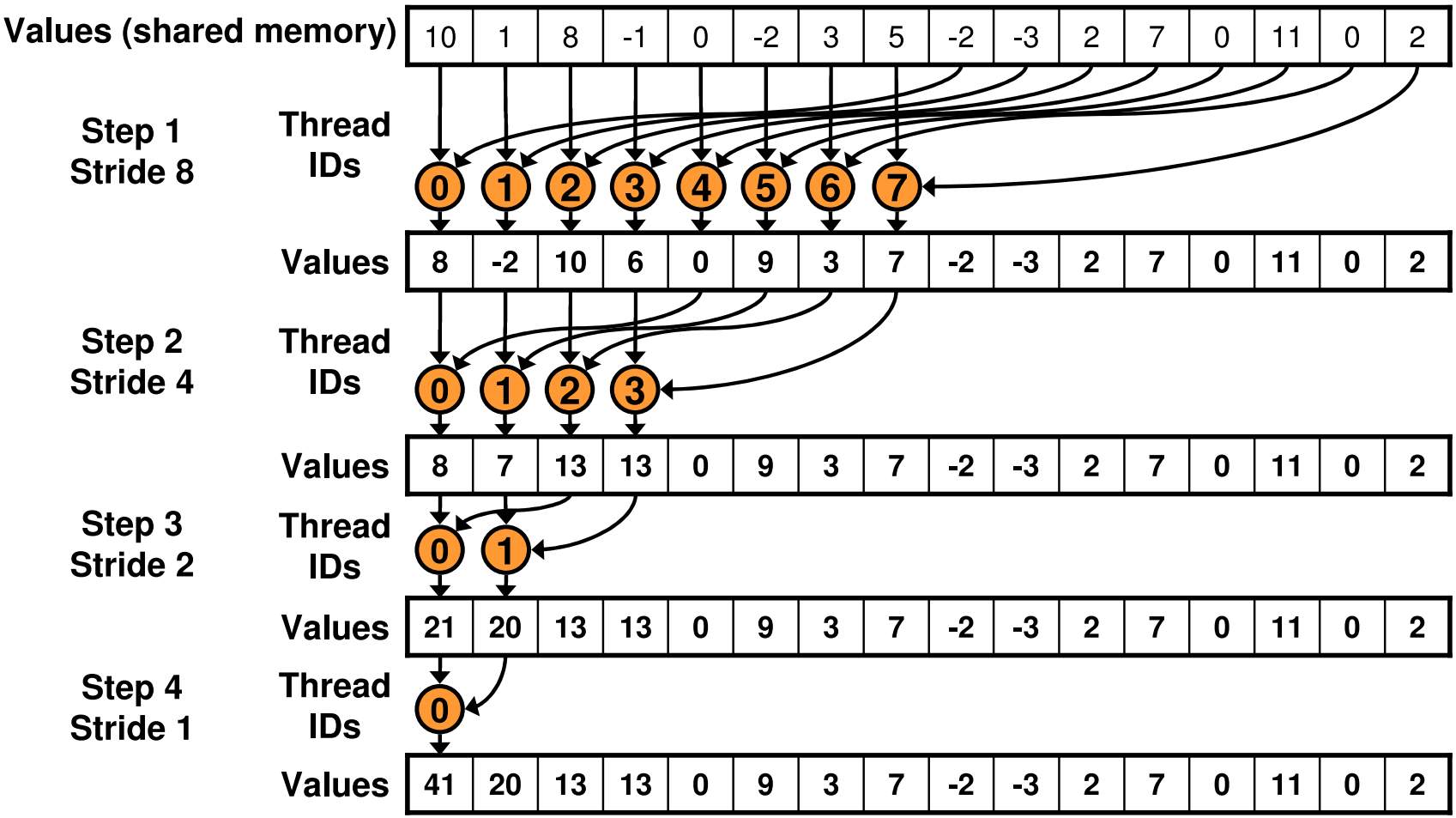
- Считаем действие в каждом узле

  Легко параллелизуется — по треду на узел.

- Суммируем

  Уже не получается параллелизовать наивно.

Алгоритм суммирования



Алгоритм суммирования

- Каждая итерация уменьшает число элементов в $N_{threads}$ раз
- Итерации:
  1. 2048×2048 элементов.
  2. 16384 элементов. Всё ещё много. Нужна ещё итерация.
  3. 64 элемента. Их уже можно просуммировать на CPU.

Алгоритм суммирования

```
__global__ void
sum(const double* g_data, double* g_out, int n) {
  __shared__ double sdata[256];
  int tid = threadIdx.x;
  int i   = blockDim.x * blockIdx.x + threadIdx.x;
  if( i < n) sdata[tid] = g_data[i];
  __syncthreads();

  for( unsigned int s = blockDim.x / 2; s > 0; s >>= 1 ) {
    if( tid < s ) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if( tid == 0 ) g_out[blockIdx.x] = sdata[0];
```

Вычисление действия

```
// Parameters for lagrangian
struct Params {
    double lambda;
    double eps;
};
struct Lattice {
    double step;
    int nX,nY;
};

__device__ __forceinline__ double
potential(double phi, const Params& pars) {
  double phi2_1 = phi * phi - 1;
  return 0.5 * pars.lambda * phi2_1 * phi2_1 -
    pars.eps * phi;
}
```

Вычисление действия

```
__global__ void action(const double* field, double* out,
                       const Lattice lat, const Params  par)
{
    __shared__ double sdata[256]; // MAGIC!
    int tid = threadIdx.x;
    int i   = blockDim.x * blockIdx.x + threadIdx.x;

    // Calculate indices
    sdata[tid] = действие в узле

    ... Далее суммируем
```

main()

```
// All initizations are complete

// Iteration 1
action<<<blocksPerGrid, threadsPerBlock>>>(
  d_field,
  d_output,
  lat, par
  );

// Iteration 2
int n = blocksPerGrid / BLOCKSIZE;
sum<<<n,threadsPerBlock>>>(
  d_output, d_output, blocksPerGrid );
```

main()

```
// Copy result from device memory to host memory
cudaMemcpy( h_out, d_out, blocksPerGrid*sizeof(double)
  cudaMemcpyDeviceToHost );
double s_gpu = 0;
for( int i = 0; i < n; i++)
  s_gpu += h_out[i];
```

Спасибо