NVIDIA CUDA И OPENACC
ЛЕКЦИЯ 5

Перепёлкин Евгений

СОДЕРЖАНИЕ

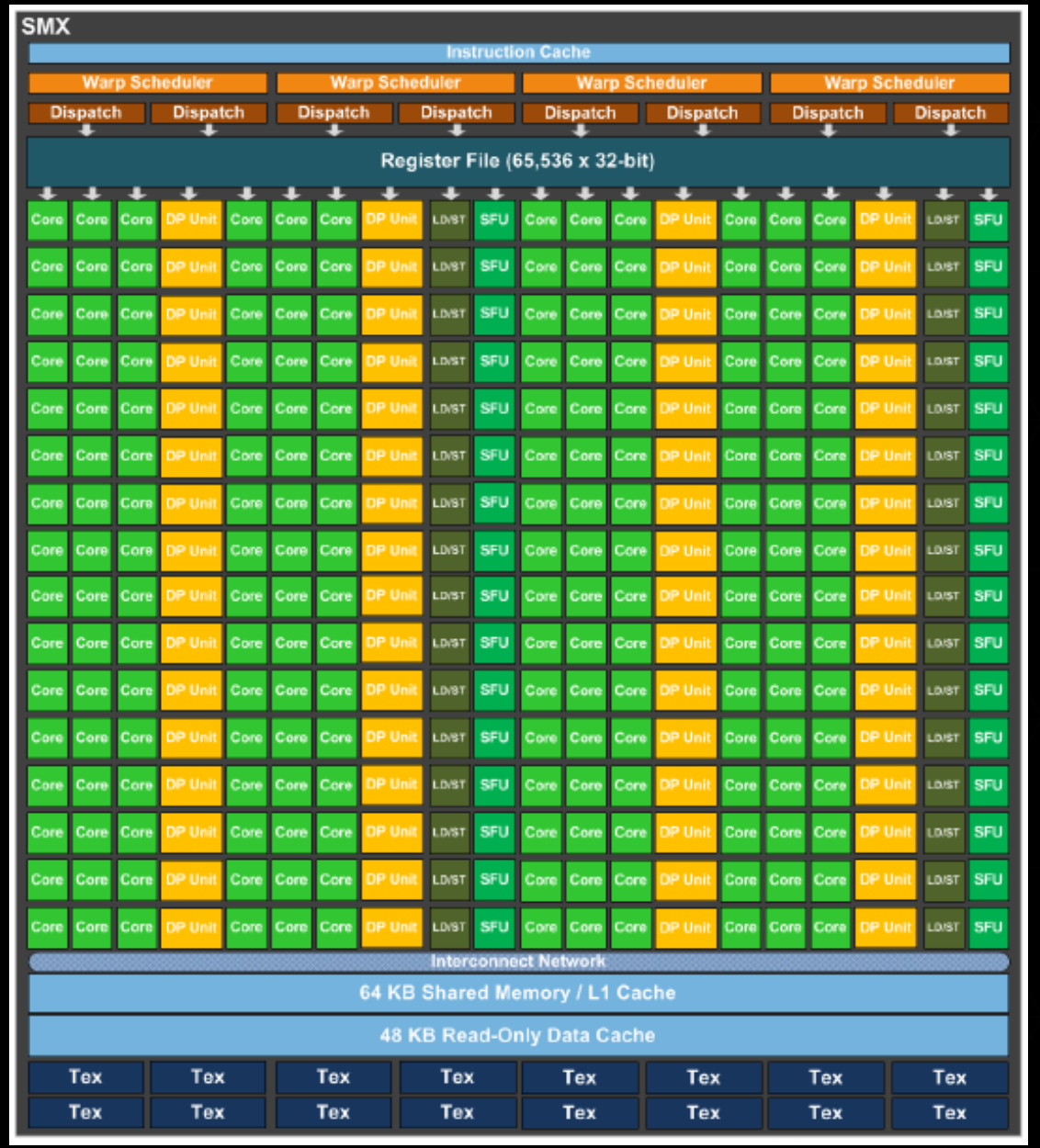
Лекция 5

- Разделяемая память
- Шаблон работы с разделяемой памятью
- Пример. Задача N-тел

Разделяемая память

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Register (регистровая)	RW	Per-thread	Высокая (on-chip)
Local (локальная)	**RW**	Per-thread	Низкая (DRAM)
Global (глобальная)	**RW**	Per-grid	Низкая (DRAM)
Shared (разделяемая)	**RW**	Per-block	Высокая (on-chip)
Constant (константная)	**RO**	Per-grid	Высокая (L1 cache)
Texture (текстурная)	**RO**	Per-grid	Высокая (L1 cache)



СТРУКТУРА SMX KEPLER

Core - 192 ядра

DP - 64 блока для вычислений с двойной точностью

SFU - 32 блока для вычислений специальных функций

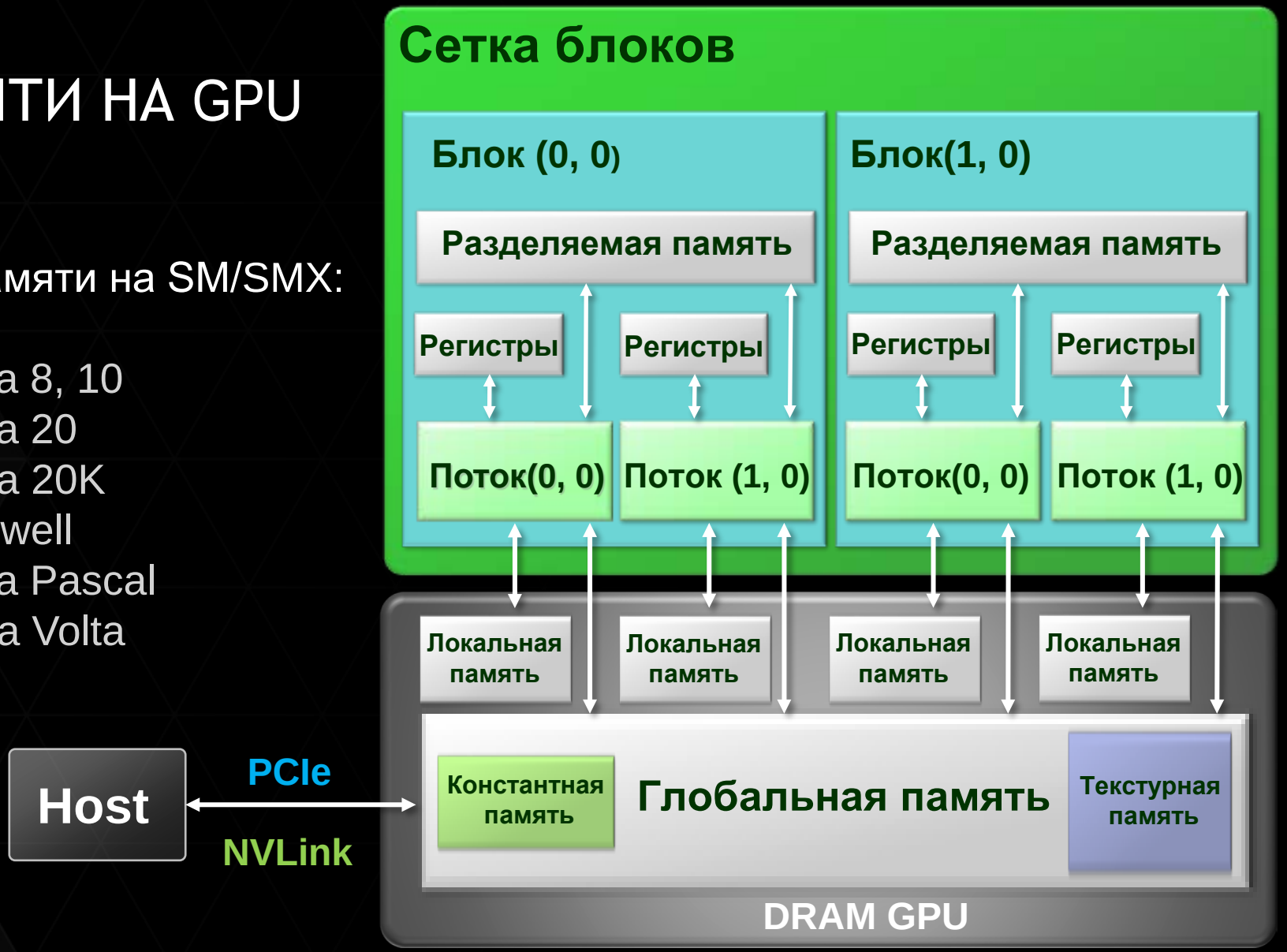
LD/ST - 32 блока загрузки / выгрузки данных

Warp Scheduler - 4 планировщика варпов

ДОСТУП К ПАМЯТИ НА GPU

Размер разделяемой памяти на SM/SMX:

- 16 КБ Tesla 8, 10
- 16/48 КБ Tesla 20
- 16/32/48 КБ Tesla 20K
- 96 КБ Maxwell
- 64 КБ Tesla Pascal
- 0/16/32/64/96 КБ Tesla Volta



РАЗДЕЛЯЕМАЯ ПАМЯТЬ

Способы выделения и синхронизация

- Статический способ:
  - `__shared__ double F [64]; // массив F`
  - `__shared__ float G;       // переменная G`
- Динамический способ:
  - `extern __shared__ float [ ];  // 64 элемента`
  - `Kernel <<< nBlock, nTread, 64 * sizeof (float) >>> (...);`
- Барьерная синхронизация:
  - `__syncthreads ()`

Шаблон работы с разделяемой памятью

ШАБЛОН

работы с разделяемой памятью

```
__global__ void My_Kernel ( float *a, float *b )
{
 int tx = threadIdx.x; // определение номера нити
 int bx = blockIdx.x; // определение номера блока
 // выделение разделяемой памяти для массива as
 __shared__ float as [BLOCK_SIZE];
 // параллельное копирование нитями блока данных из
 // массива a, расположенного в глобальной памяти в
 // массив as, расположенного в разделяемой памяти
 as [tx] = a [tx + bx * BLOCK_SIZE];
 __syncthreads (); // барьерная синхронизация нитей одного блока
 {...} // вычисление величины res, связанное с элементами массива as
 b [tx + bx * BLOCK_SIZE] = res; // параллельная запись в глобальную память
 __syncthreads (); // барьерная синхронизация нитей одного блока
}
```

Пример. Задача N-тел

$$\vec{a}_{n,i} = \frac{\vec{F}_{n,i}}{m} = Gm \sum_{k \neq n}^{N-1} \frac{\vec{r}_{k,i} - \vec{r}_{n,i}}{\left|\vec{r}_{k,i} - \vec{r}_{n,i}\right|^3},$$

$$\vec{v}_{n,i+1} = \vec{v}_{n,i} + \vec{a}_{n,i}\tau,$$

$$\vec{r}_{n,i+1} = \vec{r}_{n,i} + \vec{v}_{n,i}\tau + \vec{a}_{n,i}\frac{\tau^2}{2},$$

$$t_i = t_0 + i\tau,$$

$$\left|\vec{r}_{k,i} - \vec{r}_{n,i}\right| < 0.01\text{м}, \vec{F}_{n,i} = 0,$$

$$mG = 10\,\text{Нм}^2/\text{кг}, \tau = 0.001c$$

ЗАДАЧА N-ТЕЛ

ПРИМЕР. КОД ПРОГРАММЫ

Часть 1. Функция Acceleration_CPU

```
// CPU – вариант. Вычисление ускорения
void Acceleration_CPU (float *X, float *Y, float *AX, float *AY,
                       int nt, int N, int id)
{float ax = 0.f; float ay = 0.f; float xx, yy, rr; int sh = (nt - 1) * N;

 for ( int j = 0; j < N; j++ )  // цикл по частицам
 {if ( j != id )                // поверка самодействия
  {xx = X[j + sh] - X[id + sh]; yy = Y[j + sh] - Y[id + sh];
   rr = sqrtf (xx * xx + yy * yy);
   if ( rr > 0.01f )            // минимальное расстояние 0.01 м
   {rr = 10.f / (rr * rr * rr); ax += xx * rr; ay += yy * rr;
   } // if rr
  } // if id
 } // for
 AX[id] = ax; AY[id] = ay;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 2. Функция Position_CPU

```
// CPU-вариант. Пересчет координат
void Position_CPU (float *X, float *Y, float *VX,
                   float *VY, float *AX, float *AY,
                   float tau, int nt, int Np, int id)
{
 int sh = (nt - 1) * Np;
 X[id + nt * Np] = X[id + sh] + VX[id] * tau + AX[id] * tau * tau * 0.5f;
 Y[id + nt * Np] = Y[id + sh] + VY[id] * tau + AY[id] * tau * tau * 0.5f;

 VX[id] += AX[id] * tau;
 VY[id] += AY[id] * tau;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 3. Функция-ядро Acceleration_GPU

```
// GPU-вариант. Расчет ускорения
__global__ void Acceleration_GPU (float *X, float *Y,
                                  float *AX, float *AY, int nt, int N)
{int id = threadIdx.x + blockIdx.x * blockDim.x;
 float ax = 0.f; float ay = 0.f; float xx, yy, rr; int sh = (nt - 1) * N;
 for ( int j = 0; j < N; j++ )      // цикл по частицам
 {if (j != id)                      // проверка самодействия
  {xx = X[j + sh] - X[id + sh]; yy = Y[j + sh] - Y[id + sh];
   rr = sqrtf (xx * xx + yy * yy);
   if (rr > 0.01f)                  // минимальное расстояние 0.01 м
   {rr = 10.f / (rr * rr * rr); ax += xx * rr; ay += yy * rr;
   } // if rr
  } // if id
 } // for j
 AX[id] = ax; AY[id] = ay;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 4. Функция-ядро Position_GPU

```
// GPU-вариант. Пересчет координат
__global__ void Position_GPU (float *X, float *Y, float *VX, float *VY,
                              float *AX, float *AY, float tau, int nt, int Np)
{
 int id = threadIdx.x + blockIdx.x * blockDim.x;
 int sh = (nt – 1) * Np;

 X[id + nt * Np] = X[id + sh] + VX[id] * tau + AX[id] * tau * tau * 0.5f;
 Y[id + nt * Np] = Y[id + sh] + VY[id] * tau + AY[id] * tau * tau * 0.5f;

 VX[id] += AX[id] * tau;
 VY[id] += AY[id] * tau;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 5. Функция main

```
int main (int argc, char* argv[])
{float timerValueGPU, timerValueCPU;
 cudaEvent_t start, stop;  // определение переменных-событий для таймера
 cudaEventCreate ( &start ); cudaEventCreate ( &stop );

 int N = 10240;          // число частиц (2-й вариант 20480)
 int NT = 10;            // число шагов по времени (для анимации - 800)
 float tau = 0.001f;  // шаг по времени 0.001 с

// создание массивов на host
 float *hX, *hY, *hVX, *hVY, *hAX, *hAY;
 unsigned int mem_size     = sizeof (float) * N;
 unsigned int mem_size_big = sizeof (float) * NT * N;
 hX  = (float*) malloc (mem_size_big); hY  = (float*) malloc (mem_size_big);
 hVX = (float*) malloc (mem_size);     hVY = (float*) malloc (mem_size);
 hAX = (float*) malloc (mem_size);     hAY = (float*) malloc (mem_size);
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 6. Функция main

```
// задание начальных условий на host
 float vv,phi;
 for ( j = 0; j < N; j++ )
 {phi = (float) rand();
  hX[j]  = rand() * cosf(phi) * 1.e-4f; hY[j] = rand() * sinf(phi) * 1.e-4f;
  vv = (hX[j] * hX[j] + hY[j] * hY[j]) * 10.f;
  hVX[j] = - vv * sinf(phi); hVY[j] = vv * cosf(phi);
 }
// создание на device массивов
 float *dX, *dY, *dVX, *dVY, *dAX, *dAY;
 cudaMalloc((void**) &dX, mem_size_big);
 cudaMalloc((void**) &dY, mem_size_big);
 cudaMalloc((void**) &dVX, mem_size); cudaMalloc((void**) &dVY, mem_size);
 cudaMalloc((void**) &dAX, mem_size); cudaMalloc((void**) &dAY, mem_size);
// задание сетки нитей и блоков
 int N_thread = 256; int N_block = N / N_thread;
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 7. Функция main

```
// -----------------GPU-вариант-----------------------------
cudaEventRecord ( start, 0 );
// копирование данных на device
cudaMemcpy ( dX, hX, mem_size_big, cudaMemcpyHostToDevice );
cudaMemcpy ( dY, hY, mem_size_big, cudaMemcpyHostToDevice );
cudaMemcpy ( dVX, hVX, mem_size, cudaMemcpyHostToDevice );
cudaMemcpy ( dVY, hVY, mem_size, cudaMemcpyHostToDevice );

for ( j = 1; j < NT; j++ )
{// расчет ускорения
 Acceleration_GPU <<< N_block, N_thread >>> ( dX, dY, dAX, dAY, j, N );
 // пересчет координат
 Position_GPU <<< N_block, N_thread >>>
                    ( dX, dY, dVX, dVY, dAX, dAY, tau, j, N );
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 8. Функция main

```
// копирование траекторий с device на host
cudaMemcpy ( hX, dX, mem_size_big, cudaMemcpyDeviceToHost );
cudaMemcpy ( hY, dY, mem_size_big, cudaMemcpyDeviceToHost );

// определение времени выполнения GPU-варианта
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &timerValueGPU, start, stop );
printf ("\n GPU calculation time %f msec\n", timerValueGPU);

{...} // сохранение траекторий в файл, GPU-вариант
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 9. Функция main

```
//------------------CPU-вариант----------------------
cudaEventRecord ( start, 0 );
for ( j = 1; j < NT; j++ )
{for ( id = 0; id < N; id++ )
 {Acceleration_CPU (hX, hY, hAX, hAY, j, N, id);
  Position_CPU (hX, hY, hVX, hVY, hAX, hAY, tau, j, N, id);
 }
}
// определение времени выполнения CPU-варианта
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &timerValueCPU, start, stop );
printf ("\n CPU calculation time %f msec\n", timerValueCPU);
printf ("\n Rate %f x\n",timerValueCPU/timerValueGPU);
{...} // сохранение траекторий, CPU-вариант
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 10. Функция main

```
// освобождение памяти
 free (hX); free (hY); free (hVX); free (hVY); free (hAX); free (hAY);
 cudaFree (dX); cudaFree (dY); cudaFree (dVX); cudaFree (dVY);

 // уничтожение переменных-событий
 cudaEventDestroy ( start );
 cudaEventDestroy ( stop );

 return 0;
}
```

ПРИМЕР. ЗАДАЧА N-ТЕЛ

ПРИМЕР. ЗАДАЧА N-ТЕЛ

РЕЗУЛЬТАТ

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

Number of particles	10240	20480
GPU calculation time	96.7 ms	215.7 ms
CPU calculation time	3442 ms	13821 ms
Rate	35 x	64 x

Время расчета GPU-варианта включает в себя:

- копирование данных с «host» на «device»;
- выполнение «функции-ядра»;
- копирование данных с «device» на «host».

КОД ПРОГРАММЫ НА CUDA

```
__global__ void Acceleration_Shared (float *X, float *Y, float *AX, float *AY,
                                     int nt, int N, int N_block)
{int id = threadIdx.x + blockIdx.x * blockDim.x;
 float ax = 0.f; float ay = 0.f; float xx, yy, rr; int sh = (nt - 1) * N;
 float xxx = X[id + sh]; float yyy = Y[id + sh];
  __shared__ float Xs[256]; __shared__ float Ys[256]; // выделение разделяемой памяти
 for ( int i = 0; i < N_block; i++ )                  // основной цикл по блокам
 {Xs[threadIdx.x] = X[threadIdx.x + i * blockDim.x + sh]; // копирование из глобальной
  Ys[threadIdx.x] = Y[threadIdx.x + i * blockDim.x + sh]; // в разделяемую память
   __syncthreads ();                                  // синхронизация
  for ( int j = 0; j < blockDim.x; j++ )              // вычислительная часть
  {if ( ( j + i * blockDim.x ) != id )
   {xx = Xs[j] - xxx; yy = Ys[j] - yyy; rr = sqrtf ( xx * xx + yy * yy );
    if ( rr > 0.01f ) {rr = 10.f /(rr * rr * rr); ax += xx * rr; ay += yy * rr;} //if
   } // if id
  } // for j
   __syncthreads ();                                  // синхронизация
 } // for i
 AX[id] = ax; AY[id] = ay;
}
```

РЕЗУЛЬТАТ

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

Number of particles	10240	20480
GPU calculation time	74.4 ms	173.6 ms
CPU calculation time	3442 ms	13821 ms
Rate	46 x	79 x

Время расчета GPU-варианта включает в себя:

- копирование данных с «host» на «device»;
- выполнение «функции-ядра»;
- копирование данных с «device» на «host».