NVIDIA CUDA И OPENACC
ЛЕКЦИЯ 3

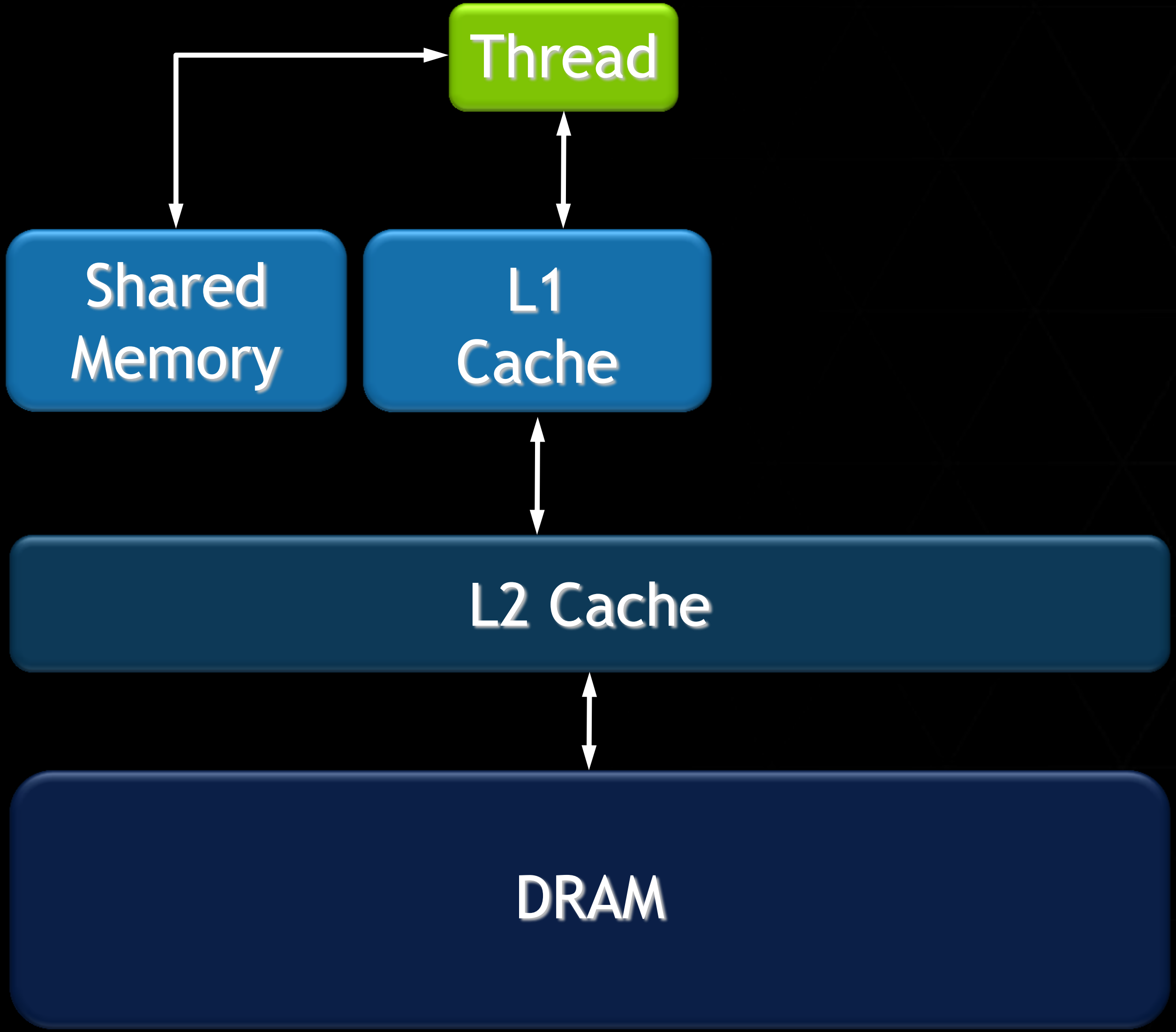
Перепёлкин Евгений

СОДЕРЖАНИЕ

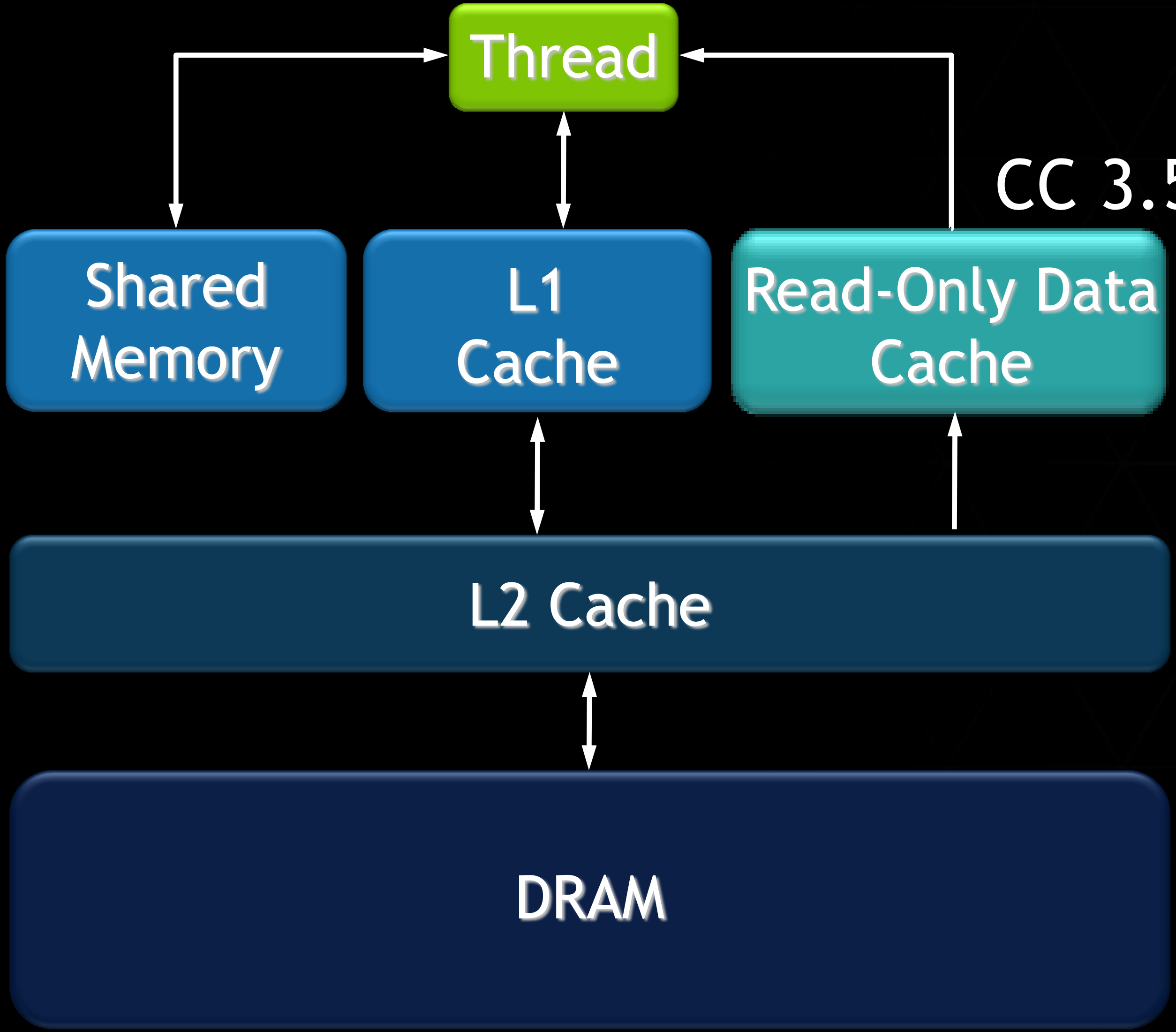
Лекция 3

- Объединение запросов в CUDA
- Пример решения СЛАУ
- Пример решения СНАУ

Объединение запросов в CUDA



ПОДСИСТЕМА ПАМЯТИ ДЛЯ CC 2.X



ПОДСИСТЕМА ПАМЯТИ ДЛЯ CC 3.X

READ-ONLY DATA CACHE

Варианты управления

Использование классификаторов `const` и `__restrict__`:

```
__global__ void kernel (int* __restrict__ output
                  const int* __restrict__ input )
{ ...
 output[idx] = input[idx];
}
```

Использование `__ldg ( )`:

```
__global__ void kernel ( int *output, int *input )
{ ...
 output[idx] = __ldg ( &input[idx] );
}
```

КОНФИГУРАЦИИ

Разделяемой памяти и L1-кэша

- 48КБ SMEM, 16КБ L1 режим: `cudaFuncCachePreferShared`
- 16КБ SMEM, 48КБ L1 режим: `cudaFuncCachePreferL1`
- Режим без предпочтения: `cudaFuncCachePreferNone`
  - В этом случае будет выбрана конфигурация в соответствии с текущим контекстом.
- По умолчанию используется конфигурация с большей разделяемой памятью: `cudaFuncCachePreferShared`
- Переключение функцией: `cudaFuncSetCacheConfig`

ШАБЛОН ВЫБОРА КОНФИГУРАЦИИ

с большим L1-кэшем

```
// device код
__global__ void My_kernel (...)
{ ... }
// host код
int main( )
{...
 cudaFuncSetCacheConfig ( My_kernel, cudaFuncCachePreferL1 );
 ...
}
```

ОБРАЩЕНИЯ В ГЛОБАЛЬНУЮ ПАМЯТЬ

Использование L1 и L2 - кэша

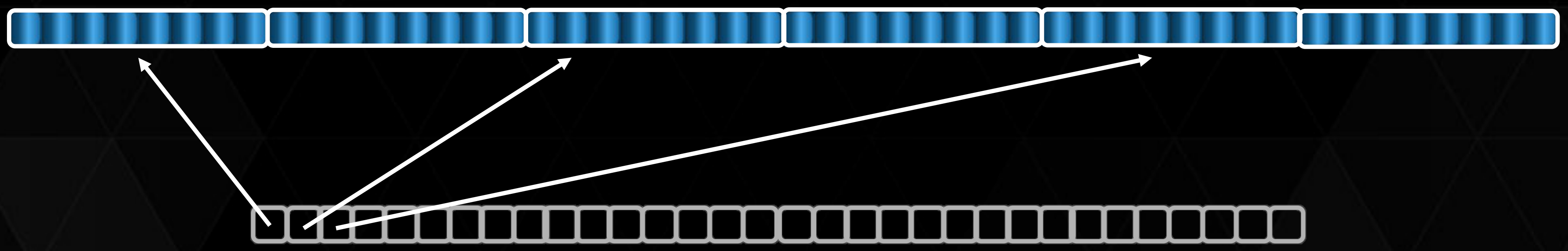
- Флаги компиляции:
  - использовать L1 и L2: -Xptxas –dlcm=ca
  - использовать L2: -Xptxas –dlcm=cg
- Кэш линия 128 Б и выравнивание по 128 Б в глобальной памяти
- Объединение запросов происходит на уровне варпов.

ОБРАЩЕНИЯ В ГЛОБАЛЬНУЮ ПАМЯТЬ

CC 2.x и выше

- L1 выключен — всегда идут запросы по 32 Б
- Лучше использовать для разряженного доступа в память

32 транзакции по 32 Б, вместо 32 транзакций по 128 Б

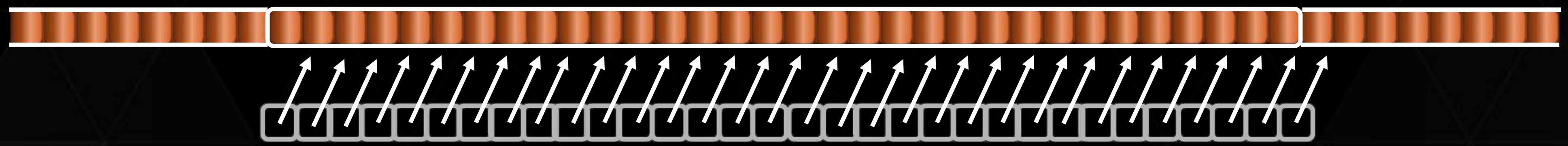


ОБРАЩЕНИЯ В ГЛОБАЛЬНУЮ ПАМЯТЬ

CC 2.x и выше

- Объединение запросов в память для 32 нитей
- L1 включен — всегда идут запросы по 128 Б с кэшированием в L1

2 транзакции — 2x 128 Б



Следующий варп скорее всего только 1 транзакция,
так как попадаем в L1

ФУНКЦИЯ

заполнения массива одинаковыми значениями

```
cudaError_t cudaMemset ( void *devPtr, int value, size_t count );
```

Пример решения СЛАУ

ПРИМЕР 1

решения системы линейных алгебраических уравнений

$$A\vec{x} = \vec{f},$$

$$x_k^{s+1} = x_k^s + \frac{1}{a_{kk}}\left(f_k - \sum_{i=1}^{N} a_{ki} x_i^s\right), \qquad (*)$$

$$1 \le k \le N, s = 0,1,2, \ldots$$

достаточное условие сходимости: $\sum_{i \ne k} \left| \frac{a_{ki}}{a_{kk}} \right| < 1$

ПРИМЕР 1

Часть 1. Функция-ядро «Solve»

```
__global__ void Solve ( double *dA, double *dF,
                        double *dX0, double *dX1, int N )
{double aa, sum = 0.;
 int t = blockIdx.x * blockDim.x + threadIdx.x;

 for ( int j = 0; j < N; j++ )
 {sum += dA [ j + t * N ] * dX0[j];
  if ( j == t ) aa = dA [ j + t * N ];
 }
  dX1[t] = dX0[t] + ( dF[t] - sum ) / aa;
}
```

ПРИМЕР 1

Обращение в память. Функция-ядро «Solve»

Для матрицы A:

```
dA [ j + t * N ]                    (*)
dA [ j ], dA [ j + N ], dA [ j + 2 * N ],...
```

Для транспонированной матрицы A^T:

```
dA [ t + j * N ]                    (**)
dA [ j * N ], dA [ 1 + j * N ], dA [ 2 + j * N ],...
```

ПРИМЕР 1

Часть 2. Функция-ядро «Eps»

```
__global__ void Eps ( double *dX0, double *dX1,
                      double *delta, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  delta[i] = abs ( dX0[i] - dX1[i] );
  dX0[i] = dX1[i];
}
```

http://nvlabs.github.io/cub/ – сайт CUB

ПРИМЕР 1

Часть 3. Фрагменты функции main

```
int main()
{...
 double EPS = 1.e-15; // Точность приближенного решения
 int N = 10240;  // Число уравнение в системе
 int size = N * N;    // Размер матрицы системы
 int N_thread = 512; // Число нитей в блоке
 unsigned int mem_sizeA = sizeof ( double ) * size; // память для матрицы
 unsigned int mem_sizeX = sizeof ( double ) * N; // память для столбцов
```

ПРИМЕР 1

Часть 4. Фрагменты функции main

```
// Выделение памяти на host

 hA =      ( double* ) malloc ( mem_sizeA ); // матрица A

 hF =      ( double* ) malloc ( mem_sizeX ); // правая часть системы F

 hX =      ( double* ) malloc ( mem_sizeX ); // точное решение

 hX0 =     ( double* ) malloc ( mem_sizeX ); // приближенное решение X(n)

 hX1 =     ( double* ) malloc ( mem_sizeX ); // приближенное решение X(n+1)

 hDelta = ( double* ) malloc ( mem_sizeX ); // разница |X(n+1)- X(n)|
```

ПРИМЕР 1

Часть 5. Фрагменты функции main

```
{ ... } // Генерация матрицы A
{ ... } // Задание точного решения и начального приближения
{ ... } // Задание правой части СЛАУ
// Выделение памяти на device
 cudaMalloc ( ( void** ) &dA, mem_sizeA );      // матрица A
 cudaMalloc ( ( void** ) &dF, mem_sizeX );      // правая часть F
 cudaMalloc ( ( void** ) &dX0, mem_sizeX );     // решение X(n)
 cudaMalloc ( ( void** ) &dX1, mem_sizeX );     // решение X(n+1)
 cudaMalloc ( ( void** ) &delta, mem_sizeX ); // разница |X(n+1)- X(n)|
```

ПРИМЕР 1

Часть 6. Фрагменты функции main

```
// ---------------------GPU вариант ---------------------

{ ... } // Задание сетки блоков

cudaEventRecord (start, 0); // Старт таймера

// Копирование данных с host на device

cudaMemcpy ( dA, hA, mem_sizeA, cudaMemcpyHostToDevice );   // матрица A

cudaMemcpy ( dF, hF, mem_sizeX, cudaMemcpyHostToDevice );   // правая часть F

cudaMemcpy ( dX0, hX0, mem_sizeX, cudaMemcpyHostToDevice ); // начальное

                                                            // приближение
```

ПРИМЕР 1

Часть 7. Фрагменты функции main

```
eps = 1.; k = 0;
while ( eps > EPS ) // Итерационный процесс
{k ++; // номер итерации
 Solve <<< N_blocks, N_thread >>> ( dA, dF, dX0, dX1, N );
 Eps <<< N_blocks, N_thread >>> ( dX0, dX1, delta, N );
 cudaMemcpy ( hDelta, delta, mem_sizeX, cudaMemcpyDeviceToHost );
 eps = 0.; for ( j = 0; j < N; j++ ) eps += hDelta[j];
 eps = eps / N; printf ("\n Eps[%i]=%e ", k, eps);
} // while
```

ПРИМЕР 1

Часть 8. Фрагменты функции main

```
// Копирование решения с device на host
 cudaMemcpy ( hX1, dX0, mem_sizeX, cudaMemcpyDeviceToHost );
// Остановка таймера и вывод времени выполнения GPU-варианта
 cudaEventRecord ( stop, 0 );
 cudaEventSynchronize ( stop );
 cudaEventElapsedTime ( &timerValueGPU, start, stop );
 printf ("\n GPU calculation time %f msec\n", timerValueGPU);
```

ПРИМЕР 1

Часть 9. Фрагменты функции main

```
while ( eps > EPS ) // итерационный процесс
 {k ++; // номер итерации
  for ( i = 0; i < N; i++ )
  {sum = 0.;
   for ( j = 0; j < N; j++ ) sum += hA[j + i * N] * hX0[j];
   hX1[i] = hX0[i] + ( hF[i] - sum ) / hA[i + i * N];
  }
  {...} // Оценка точности решения
 } // while
```

РЕЗУЛЬТАТ 1

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

```
CPU calculation time       : 4876 ms
GPU calculation time* A    : 1963 ms
GPU calculation time* A^T  : 598 ms
GPU calculation time**A    : 1570 ms
GPU calculation time** A^T : 208 ms
Rate*  A : 2.5 x, Rate*  A^T: 8.2 x
Rate** A : 3.1 x, Rate** A^T: 23.4 x
```

*время выполнения с учетом копированием данных «host» - «device»

**время выполнения ТОЛЬКО «функции-ядра»

Пример решения СНАУ

ПРИМЕР 2

Непрерывный аналог метода Ньютона (НАМН)

$$\vec{\varphi}(\vec{x}) = A(\vec{x})\vec{x} - \vec{f},$$

$$\vec{x}^*\colon\ \vec{\varphi}(\vec{x}^*) = 0$$

$$\vec{x}(t) \to \vec{x}^*, \vec{\varphi}\big(\vec{x}(t)\big) \to \vec{\varphi}(\vec{x}^*) = 0, t \to +\infty,$$

$$\frac{d}{dt}\vec{\varphi}\big(\vec{x}(t)\big) = L\big(\vec{x}(t)\big)\vec{v}(t) = -\vec{\varphi}\big(\vec{x}(t)\big), \qquad (*)$$

$\vec{v}(t) = \vec{x}_t(t), L$ – производная Фреше (оператор)

ПРИМЕР 2

Итерационная процедура

$t_n = n\tau$, где τ – шаг по "времени"

$$\vec{x}_n = \vec{x}(t_n), \vec{v}_n = \vec{v}(t_n),$$

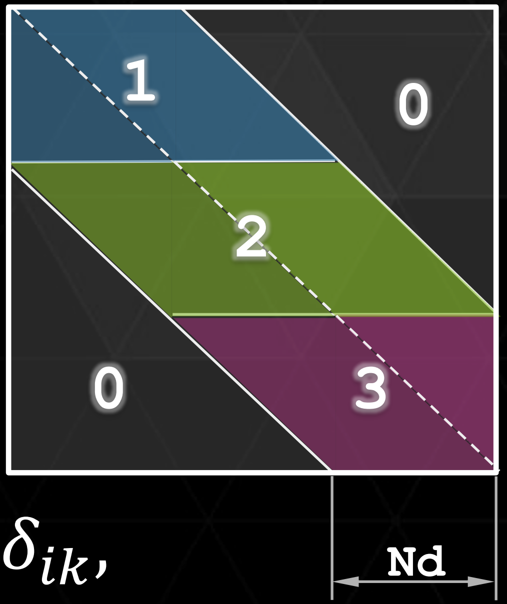
$$\vec{x}_{n+1} = \vec{x}_n + \tau\vec{v}_n,$$

$$L(\vec{x}_n)\vec{v}_n = -\vec{\varphi}(\vec{x}_n) \quad \text{– СЛАУ} \qquad (**)$$

$\vec{v}_n$ – решение СЛАУ

ПРИМЕР 2

Исходные данные



$$\varphi_i(\vec{x}) = \sum_{k=1}^{N} a_{ik}(\vec{x})x_k - f_i\,,$$

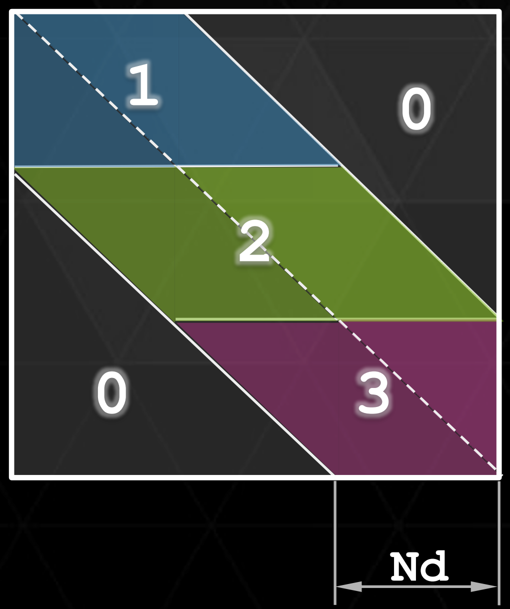
$$(***)\quad A(\vec{x}) = \{a_{ik}(\vec{x})\} = sin^2(x_i)cos^2(x_k) + N\delta_{ik},$$

$$L(\vec{x}) = \{l_{ij}(\vec{x})\} = \sum_{k=1}^{N}\left[\frac{\partial a_{ik}(\vec{x})}{\partial x_j}x_k + a_{ik}(\vec{x})\delta_{kj}\right],$$

$$\frac{\partial a_{ik}}{\partial x_j} = sin(2x_i)cos^2(x_k)\delta_{ij} - sin(2x_k)sin^2(x_i)\delta_{kj}$$

ПРИМЕР 2

Исходные данные



Точность: `double`

Количество уравнений: `N = 2048`

Погрешность для СНАУ: $\varepsilon_G = 10^{-6}$

Погрешность для СЛАУ: $\varepsilon_L = 10^{-15}$

Число ненулевых диагоналей в матрице: `Nd = (int)(0.15*N)`

«Функции-ядра»:

- **Matrix_A** - вычисление матрицы A в точке $\vec{x}$, то есть $A(\vec{x})$
- **AX** - вычисление $A(\vec{x})\vec{x}$
- **PHI** - вычисление $\varphi(\vec{x}) = A(\vec{x})\vec{x} - \vec{f}$
- **D_PHI** - вычисление производной Фреше $\mathrm{L}(\vec{x})$
- **Solve_L** - вычисление приближения $\vec{v}_n$ (СЛАУ)
- **Eps_L** - оценка погрешности решения СЛАУ
- **Solve_G** - вычисление приближения $\vec{x}_{n+1}$ по $\vec{x}_n$
- **Eps_G** - оценка погрешности решения СНАУ

ПРИМЕР 2

Функция-ядро «Matrix_A»

```
__global__ void Matrix_A ( double *dA, double *dX, int N )
{
 int j =  blockIdx.x * blockDim.x + threadIdx.x;
 int i =  blockIdx.y * blockDim.y + threadIdx.y;

 int Nd = (int)(0.15*N);

 if ( i <= j + Nd && i >= j - Nd )
 {dA[i+j*N] = pow(sin(dX[j])*cos(dX[i]),2.)+(double)N*D(i,j);
 } else
 {dA[i+j*N] = 0.;  // AT !
 }
}
```

ПРИМЕР 2

Функция-ядро «AX»

```
__global__ void AX (double *dAX, double *dA, double *dX, int N)
{
 int i =  blockIdx.x * blockDim.x + threadIdx.x;
 double sum = 0.;

 for ( int j = 0; j < N; j++ ) sum += dA[i+j*N]*dX[j]; // AT !

 dAX [i] = sum;
}
```

ПРИМЕР 2

Функция-ядро «PHI»

```
__global__ void PHI ( double *dPhi, double *dAX, double *dF )
{
 int i =  blockIdx.x * blockDim.x + threadIdx.x;

 dPhi[i] = dAX[i] - dF[i];
}
```

ПРИМЕР 2

Функция-ядро «D_PHI»

```
__global__ void D_PHI (double *dL, double *dX0, int N)
{int j =  blockIdx.x * blockDim.x + threadIdx.x;
 int i =  blockIdx.y * blockDim.y + threadIdx.y;
 double sum1 = sum2 = 0.; int k,k1,k2,Nd = (int)(0.15*N);
 if ( i <= j + Nd && i >= j - Nd )
 {if ( i >= 0 && i <= Nd ) {k1 = 0; k2 = i+Nd + 1;}      // область 1
  if ( i >= Nd+1 && i < N-Nd ) {k1 = i-Nd; k2 = i+Nd+1;} // область 2
  if ( i >= N-Nd && i < N ) {k1 = i-Nd; k2 = N;}         // область 3
  for ( k = k1; k < k2; k++ ){
  sum1 += D(k,j)*(pow(sin(dX0[i])*cos(dX0[k]),2.)+D(i,k)*(double)N);
  sum2 += dX0[k]*(sin(2.*dX0[i])*pow(cos(dX0[k]),2.)*D(i,j)-
          sin(2.*dX0[k])*pow(sin(dX0[i]),2.)*D(k,j));} // k
  dL[i+j*N] = sum1 + sum2;  // dLT !
 } else {dL[i+j*N] = 0.;}
}
```

ПРИМЕР 2

Функции-ядра «Solve_G» и «Eps_G»

```
__global__ void Solve_G (double *dX0, double *dX1, double *dV0,
                         double tau)
{
 int i =  blockIdx.x * blockDim.x + threadIdx.x;
 dX1[i] = dX0[i] + tau * dV0[i];
}

__global__ void Eps_G (double *dX0, double *dX1, double *d_dX )
{
 int i =  blockIdx.x * blockDim.x + threadIdx.x;
 d_dX[i] = abs (dX0[i] - dX1[i]);
 dX0[i] = dX1[i];
}
```

ПРИМЕР 2

Фрагмент функции «main»

```
while ( eps_G > EPS_G )
{Matrix_A <<< nBlk_MtrxA, nTid_MtrxA >>> ( dA, dX0, N );
 AX <<< nBlk_L, nTid_L >>> ( dAX, dA, dX0, N );
 PHI <<< nBlk_L, nTid_L >>> ( dPhi, dAX, dF );
 D_PHI <<< nBlk_MtrxA, nTid_MtrxA >>> ( dL, dX0, N );
 cudaMemset ( dV0, 1, mem_sizeX ); eps_L = 1.;
 while ( eps_L > EPS_L )
 {Solve_L <<< nBlk_L, nTid_L >>> ( dL, dPhi, dV0, dV1, N );
  Eps_L <<< nBlk_L, nTid_L >>> ( dV0, dV1, d_dV, N );
  cudaMemcpy ( h_dV, d_dV, mem_sizeX, cudaMemcpyDeviceToHost );
  eps_L=0.; for ( j = 0; j < N; j++ ) eps_L += h_dV[j]; eps_L = eps_L / N;
 } //while_L
 Solve_G <<< nBlk_L, nTid_L >>> ( dX0, dX1, dV0, tau );
 Eps_G <<< nBlk_L, nTid_L >>> ( dX0, dX1, d_dX );
 cudaMemcpy ( h_dX, d_dX, mem_sizeX, cudaMemcpyDeviceToHost );
 eps_G=0.; for ( k = 0; k < N; k++ ) eps_G += h_dX[k]; eps_G = eps_G / N;
} //while_G
```

РЕЗУЛЬТАТ 2

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

```
CPU calculation time : 270473 ms
GPU calculation time*: 8860 ms
Rate                 : 30 x
```

*время выполнения с учетом копирования данных «host» - «device»