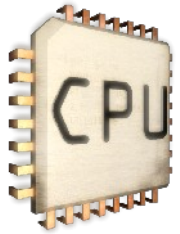
GPU версия CFD пакета SigmaFlow: портирование и оптимизация с использованием инструментария TTG Apptimizer

Авторы:
Гаврилов А.А.,
Кривов М.А.,
Гризан С.А.,
Дектерёв А.А.

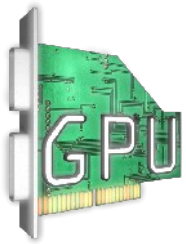
Челябинск, 2013

Распределение ролей

- Разработка пакета SigmaFlow (1993 — н.м.)
  - Красноярский филиал института Теплофизики СО РАН
  - Кафедра теплофизики СФУ
  - ООО «ТОРИНС»



- Портирование пакета SigmaFlow на GPU (2012-н.м.)
  - ООО «ТТГ Лабс»



Детали в «Суперкомьютерах»

Страницы 50 - 53

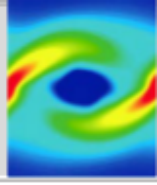
Схема выступления

- О пакете SigmaFlow
- Этап 1. Подготовка
- Этап 2. Портирование
- Этап 3. Оптимизация
- Результаты

Схема выступления

- **О пакете SigmaFlow**
- Этап 1. Подготовка
- Этап 2. Портирование
- Этап 3. Оптимизация
- Результаты

σFlow: моделируемые физико-химические процессы



• пространственные стационарные и нестационарные ламинарные и турбулентные течения;

• процессы смешения и диффузии неоднородных газовых смесей;

• химические реакции в потоке, горение газообразного, жидкого и твердого топлива;

• конвективный, радиационный теплообмен,теплопроводность;

• движение дисперсной фазы (твердые частицы, капли) в потоке газа;

• процессы сушки, пиролиза и горения частиц дисперсной фазы;

• процессы с фазовыми переходами (кавитация, кристаллизация);

• течения со свободной поверхностью;

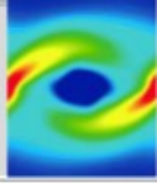
• течения со взаимодействием газа с подвижными твердыми телами;

• течения с объемными силами;

• деформация упругой среды;

• аэроупругость.

Математическая формулировка



уравнение неразрывности

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{v}) = 0$$

уравнения баланса количества движения

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla(\rho \mathbf{v} \cdot \boldsymbol{\tau}) = -\nabla p + \nabla(\boldsymbol{\tau}_m + \boldsymbol{\tau}_t) + \mathbf{g}(\rho - \rho_\infty) + \mathbf{f}$$

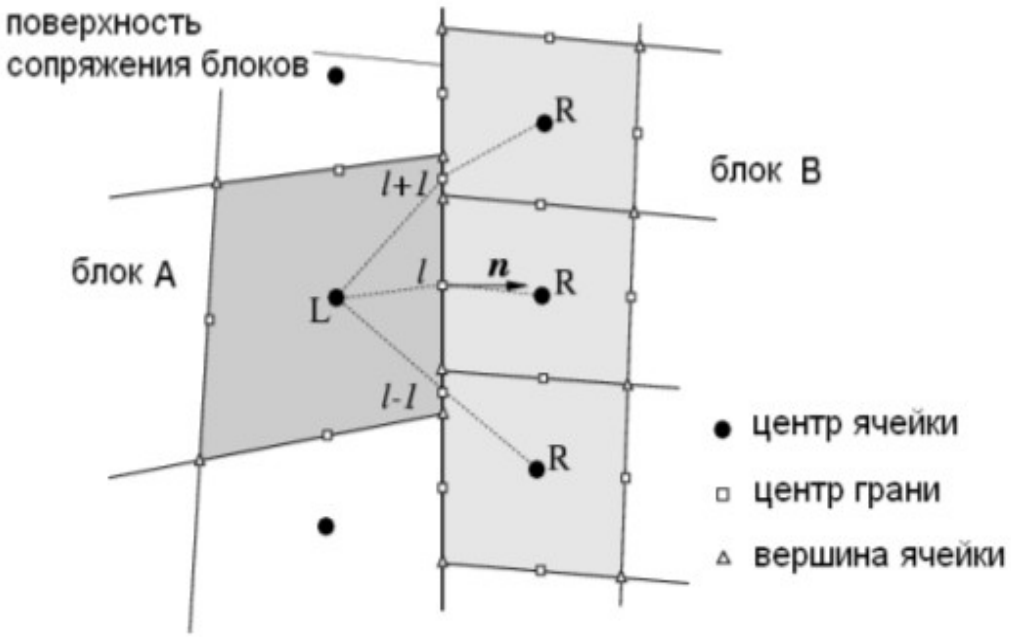
уравнение переноса энергии

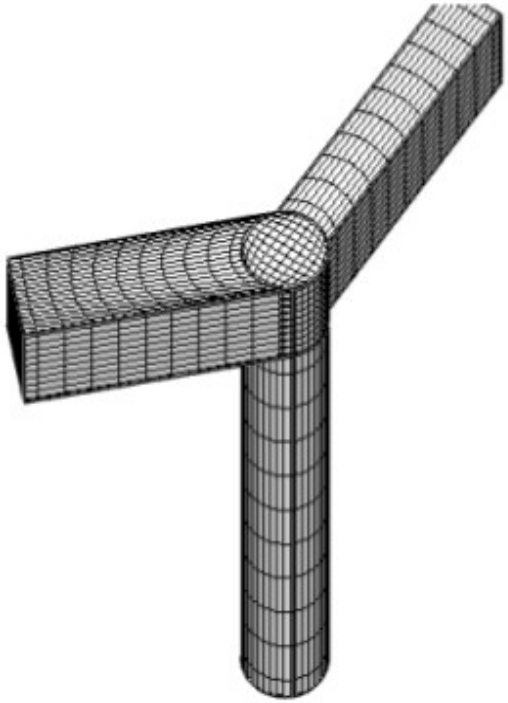
$$\frac{\partial \rho h}{\partial t} + \nabla(\rho \mathbf{v} \cdot h) = \nabla\left((\lambda + \frac{c_p \mu_t}{Pr_t}) \cdot \nabla T\right) + Q$$

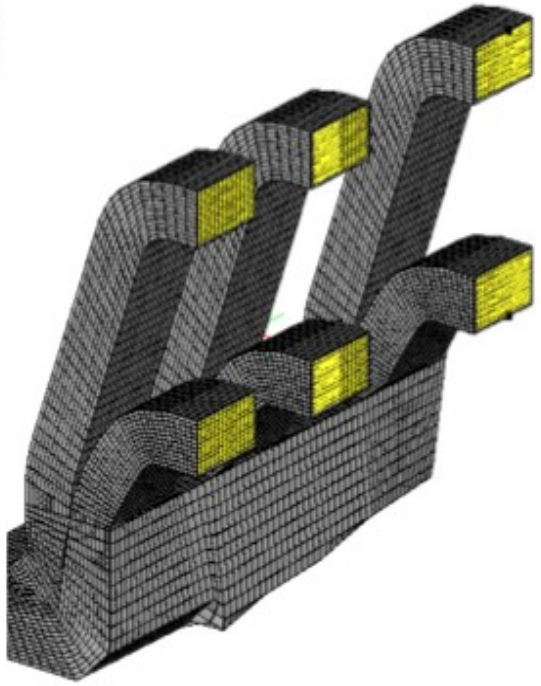
уравнение переноса концентрации (массовой доли) i-го компонента

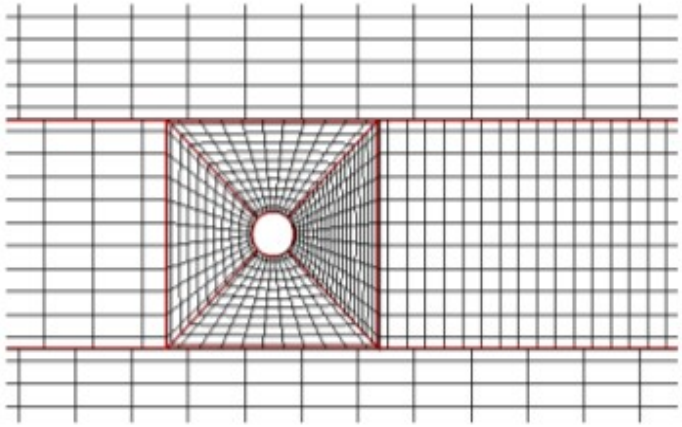
$$\frac{\partial \rho f_i}{\partial t} + \nabla(\rho \mathbf{v} \cdot f_i) = \nabla\left((D_i + \frac{\mu_t}{Sc_i}) \cdot \nabla f_i\right) + S_i$$

Расчетные сетки

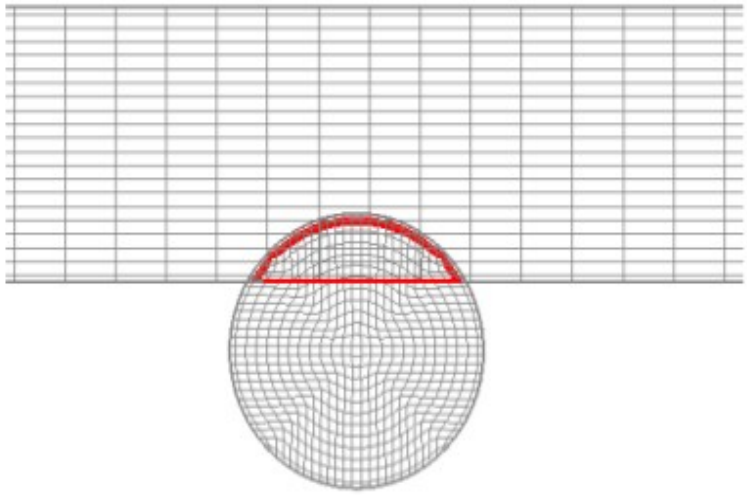




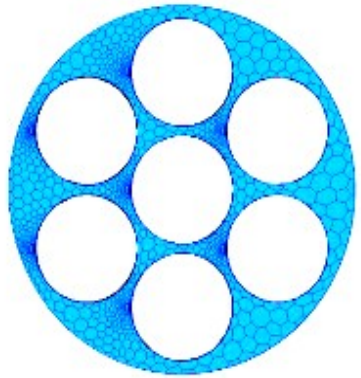




несогласованные многоблочные сетки



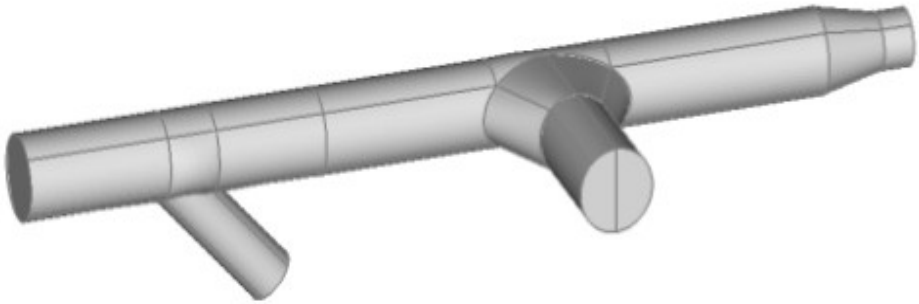
перекрывающиеся сетки

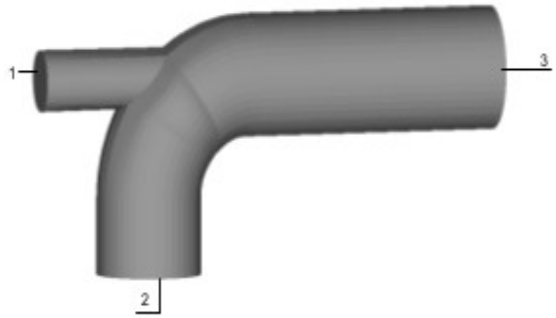


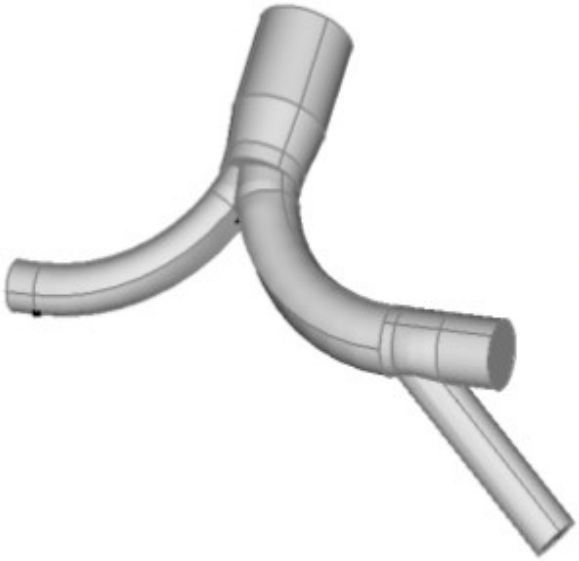
полиэдральные сетки

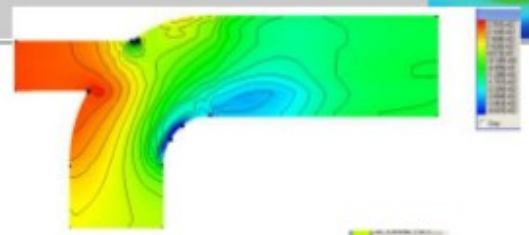
σFlow

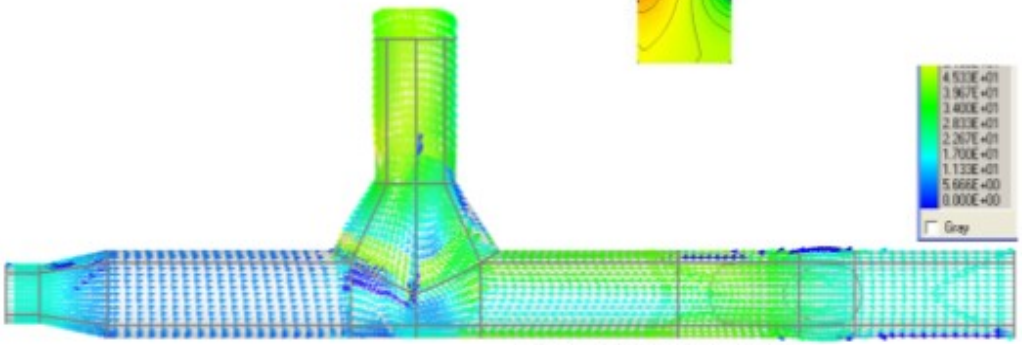
ГАЗОХОДЫ





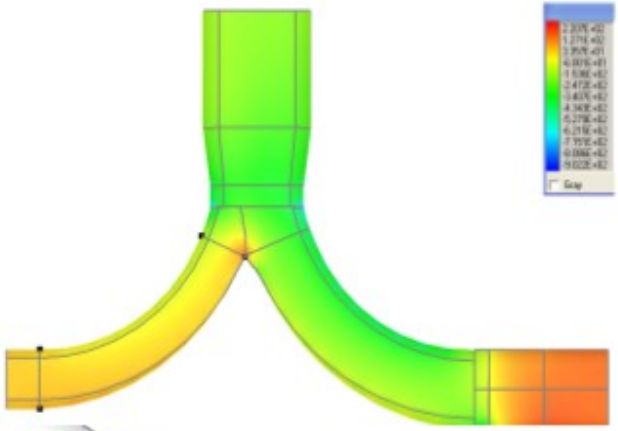


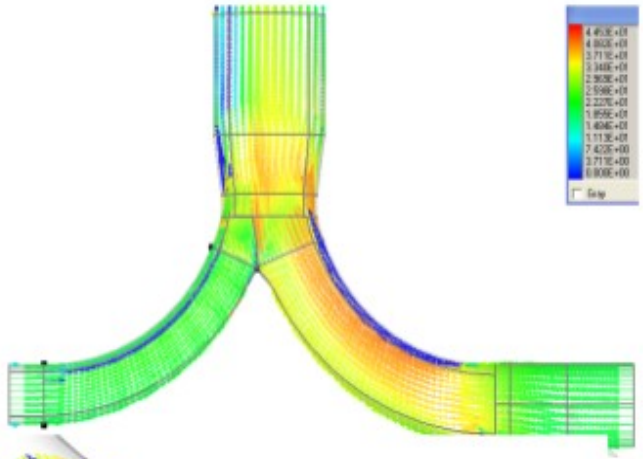


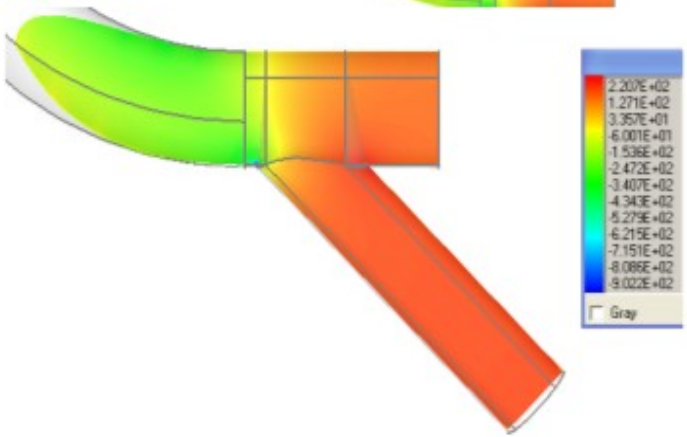


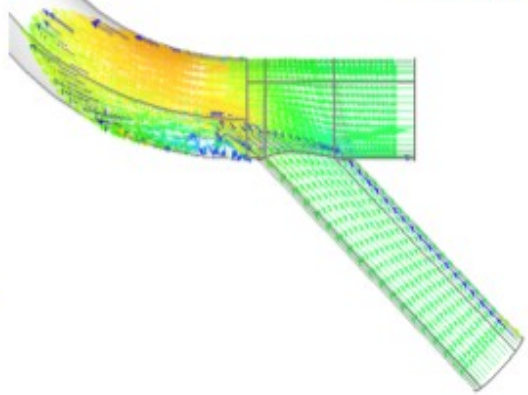
Поле давления

Поле скорости



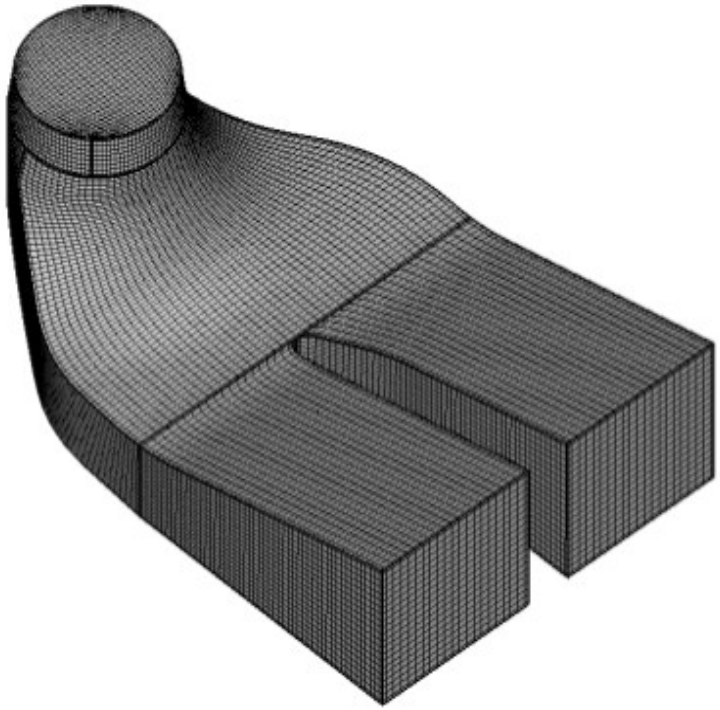




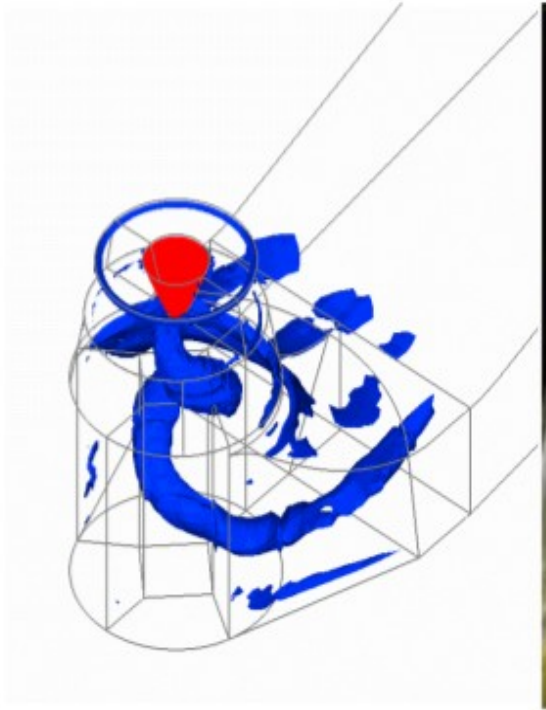


low

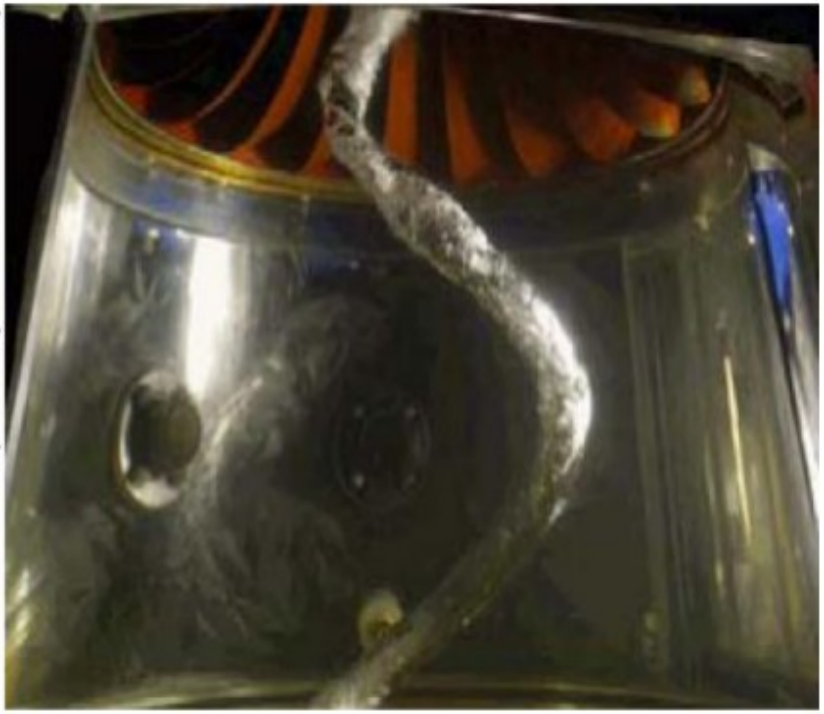
Расчет течения в отсасывающей трубе Усть-Ильимской ГЭС

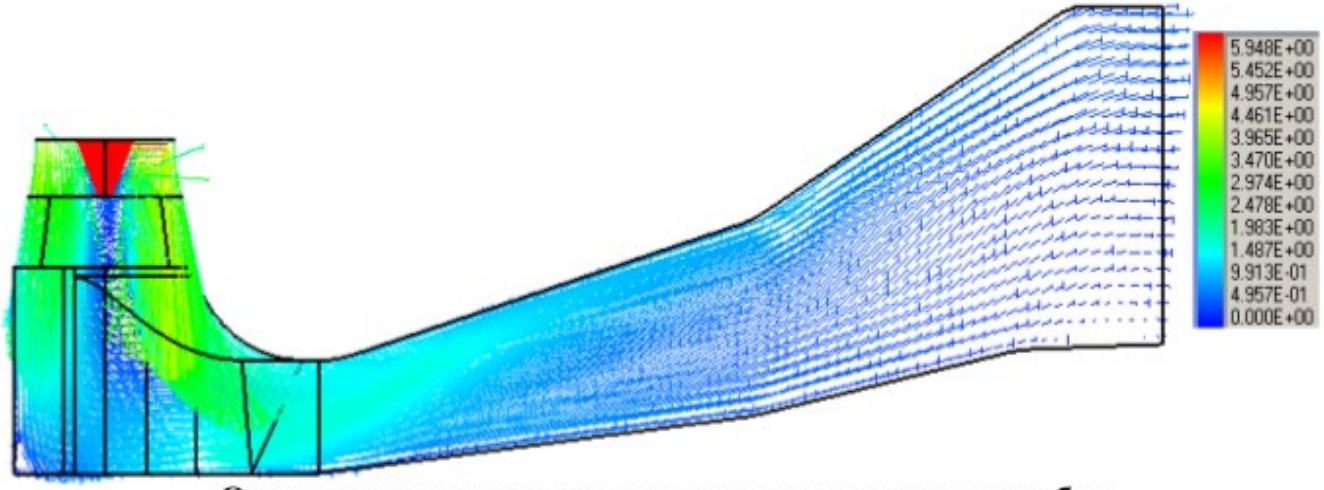


Расчетная сетка

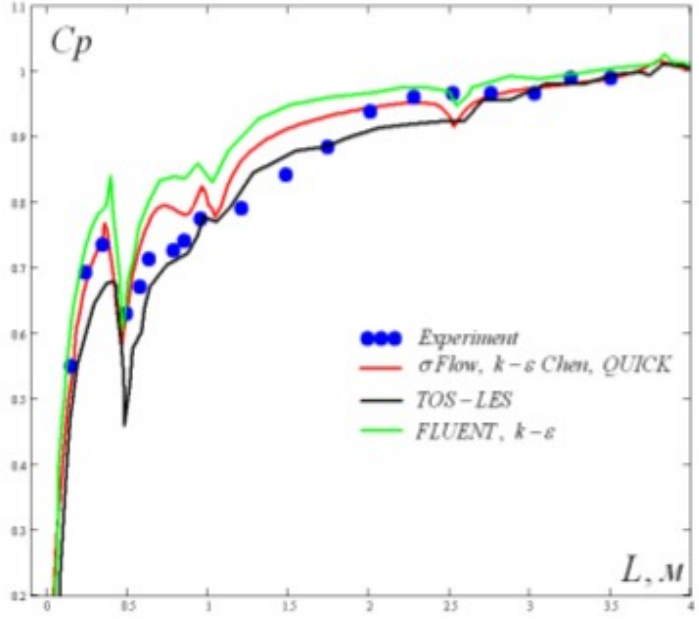


Прецессия вихревого ядра

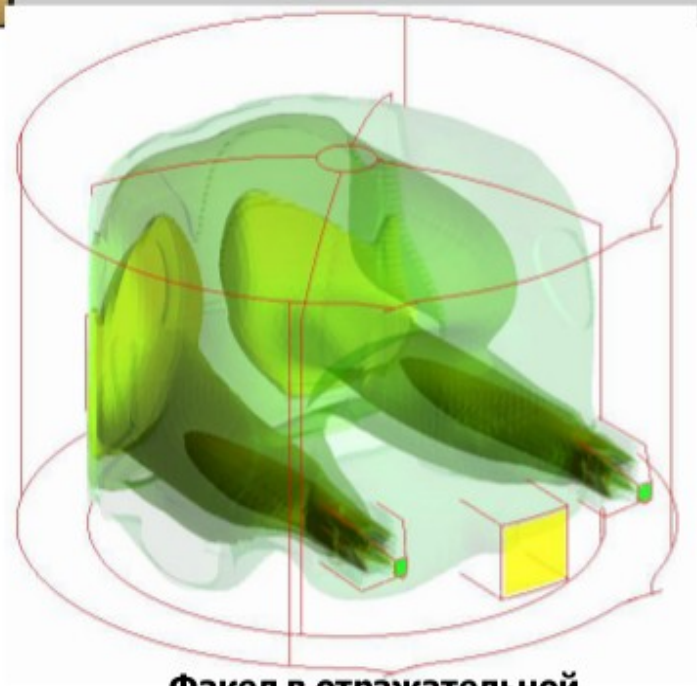




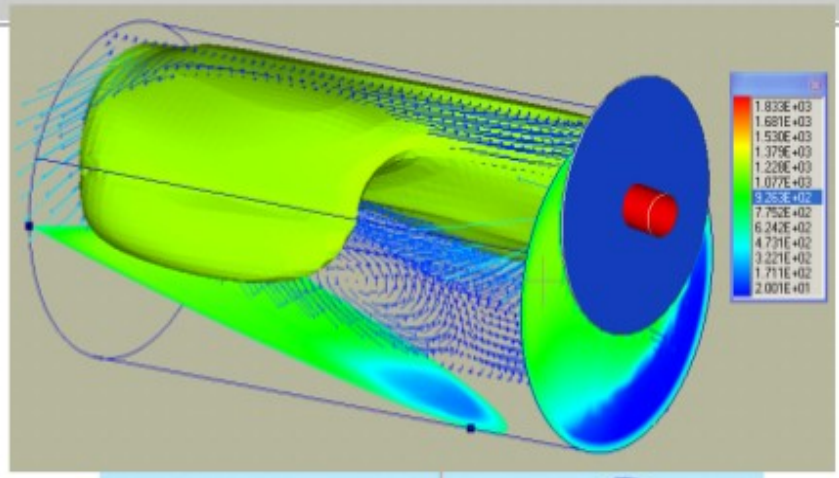
Осредненное поле вектора скорости сечении трубы

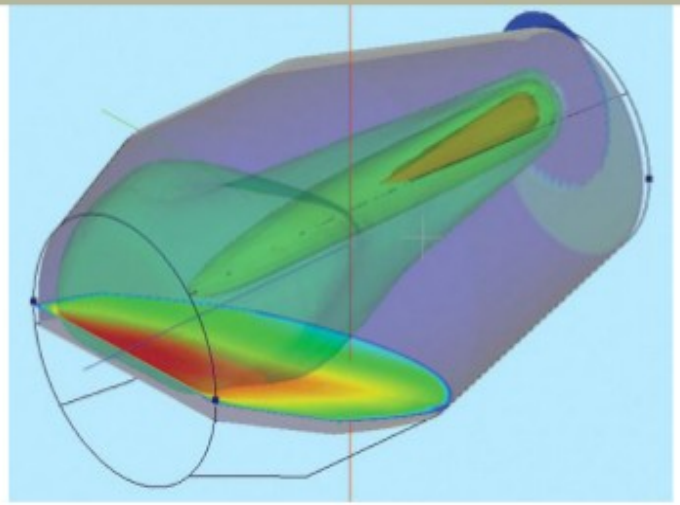


МЕТАЛЛУРГИЧЕСКИЕ ПЕЧИ

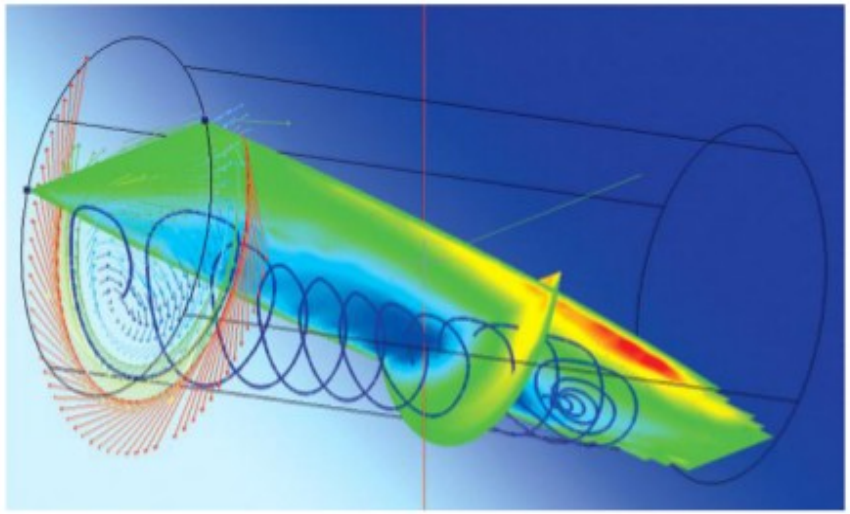


Факел в отражательной печи

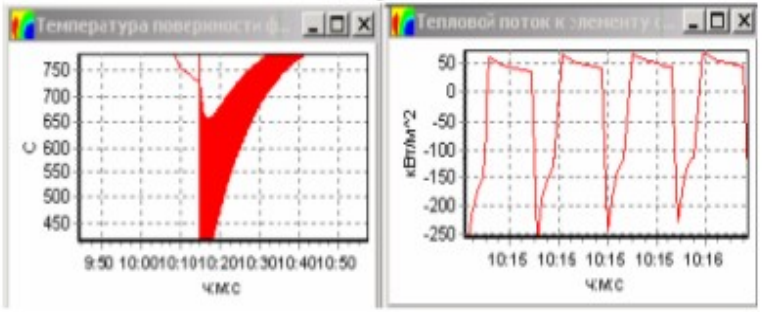




Температура факела и поверхности шихты в роторной печи



Движение металла в роторной печи



σFlow

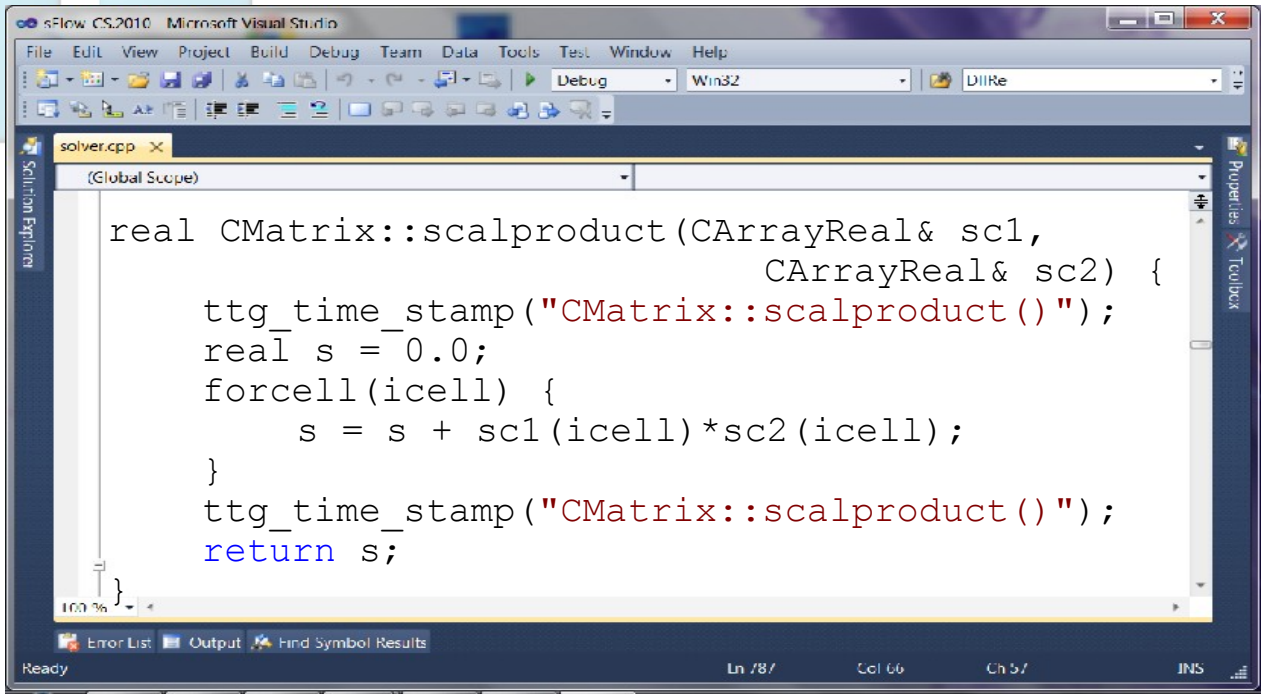
Схема выступления

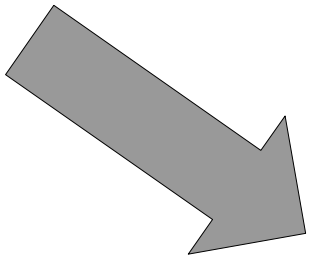
- О пакете SigmaFlow
- **Этап 1. Подготовка**
- Этап 2. Портирование
- Этап 3. Оптимизация
- Результаты

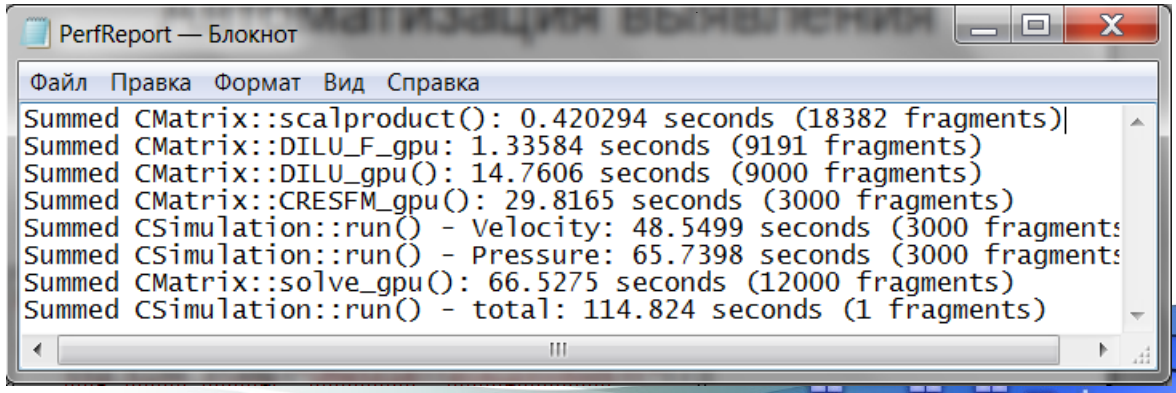
Автоматизация выявления «узких» мест

- **Проблема:** в зависимости от типа сеток и алгоритмов «узкие» места перестают быть «узкими», и наоборот
  - На одной сетке «узким» местом является предобуславливатель, а на другой сетке — скалярное произведение
  - Профайлеры типа Intel Parallel Studio оказываются малоприменимы
- **Решение:** использование временных меток
  - Возможность учитывать реальную структуру программы
  - Возможность нужной автоматизации процесса профилировки
  - Учёт как CPU, так и GPU специфики

Автоматизация выявления «узких» мест







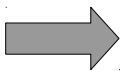
Автоматизация проверки корректности

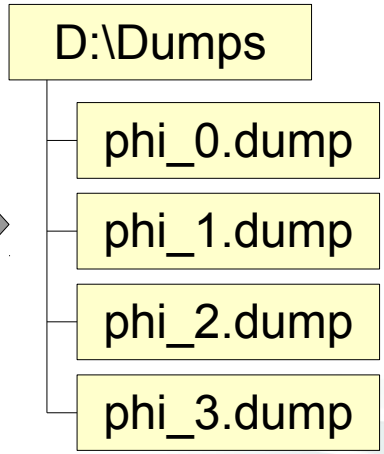
- Проблема: как понять, что в результате портирования на GPU программа не «сломалась»?
- Идея решения: сравнивать результаты с эталонными (так называемой gold-версией)
  - Не подходит, т. к. из-за большого объёма вычислений накапливается существенная погрешность, вносимая:
    - Атомарными операциями
    - Иным порядком редукции массивов
    - Другой реализацией некоторых GPU-ядер

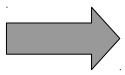
Автоматизация проверки корректности

- Решение: сравнивать промежуточные данных
  - Шаг 1. Запускается исходная версия программы, которая выгружает все нужные промежуточные массивы в dump-файлы
  - Шаг 2. Запускается изменённая версия программы, которая сравнивает текущие значения с эталонными по нормам L2 и C
  - Шаг 3. Подготавливается отчёт о найденных отличиях


```
for (int iter = 0;
      iter < 4; i++) {
    ttg_dump(«phi», phi);
    solve_cpu(phi);
}
```








```
for (int iter = 0;
      iter < 4; i++) {
    ttg_dump(«phi», phi);
    solve_gpu(phi);
}
```

Пример автоматизации проверки корректности

- Отличия промежуточных данных в CPU и GPU версиях пакета SigmaFlow в зависимости от номера итерации

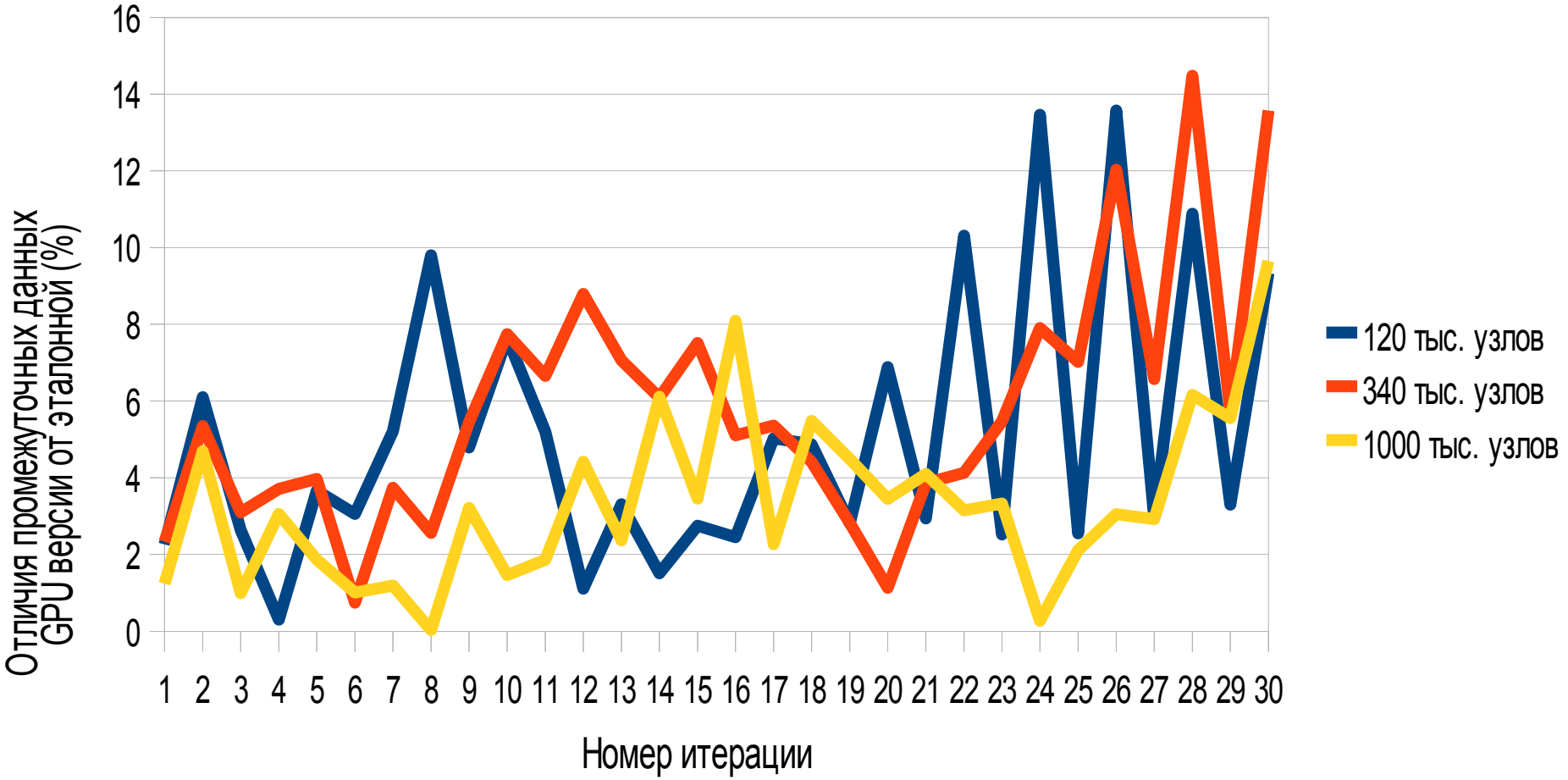
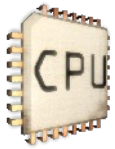


Схема выступления

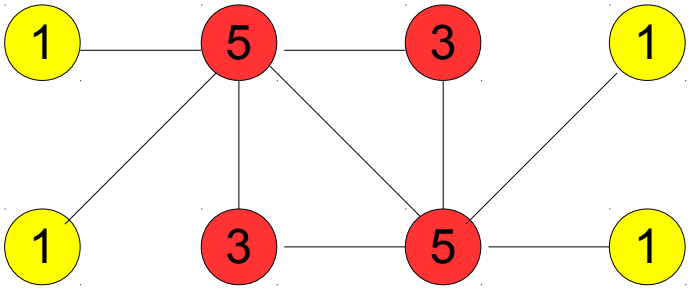
- О пакете SigmaFlow
- Этап 1. Подготовка
- **Этап 2. Портирование**
- Этап 3. Оптимизация
- Результаты

Проблема 1: гонка данных

- В некоторых ядрах обход графа, соответствующего сетке, осуществляется по рёбрам




```
for (int i=0; i<bounds.size();i++)
{
    cells[bounds[i].s] += f(i);
    cells[bounds[i].r] += g(i);
}
```



При переносе подобного кода на GPU происходит гонка данных при записи значений в массив cells.

Проблема 1: гонка данных

- Решение 1: использование атомарных операций

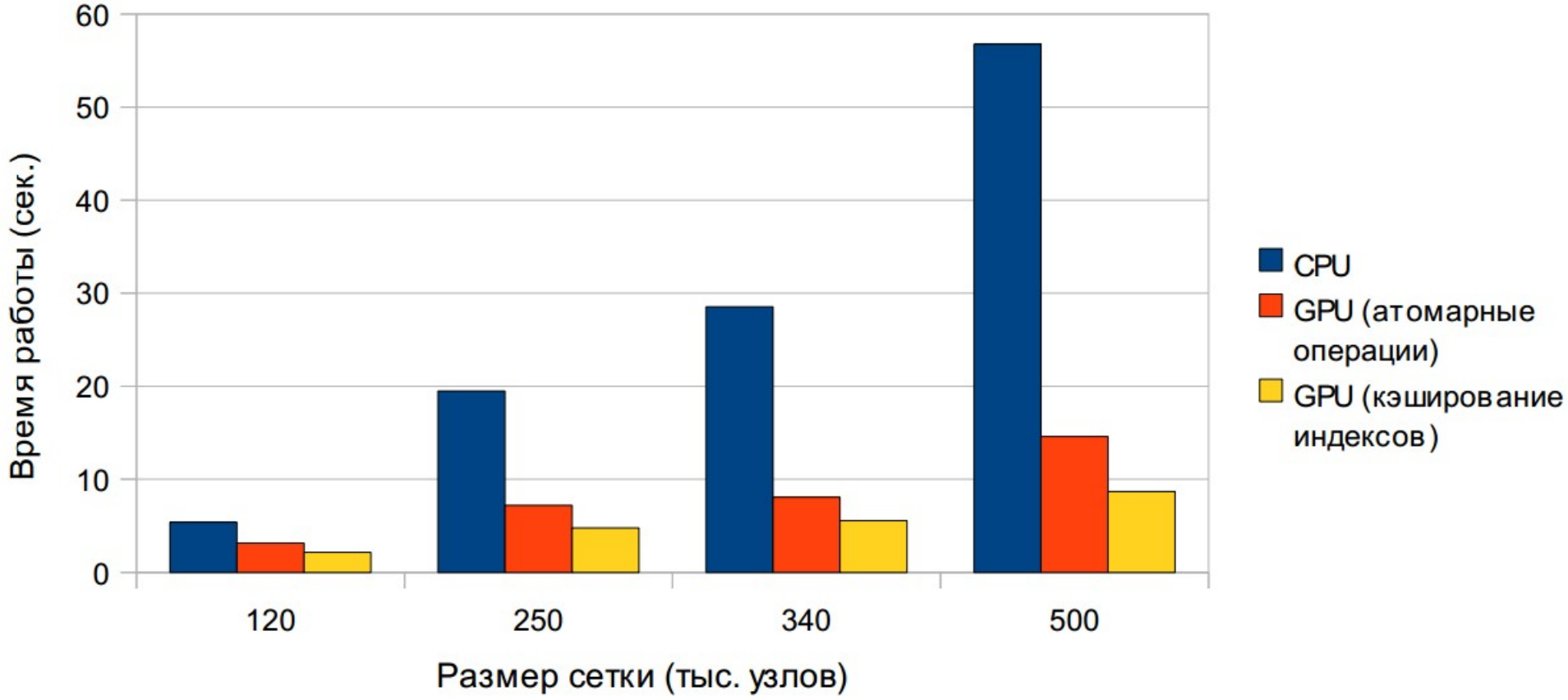

```
int i = threadIdx.x + blockIdx.x*blockDim.x;
if (int i < bounds_size)
{
   atomicAdd(&cells[bounds_s[i], f(i));
   atomicAdd(&cells[bounds_r[i]], g(i));
}
```

- Решение 2: использование вспомогательного массива с индексами и обход графа по вершинам


```
int i = threadIdx.x + blockIdx.x*blockDim.x;
if (int i < cells_size)
{
   real cell = cells[i];
   for (int j=0; j<conn_size;j++)
      cell += h(conn[j]);
   cells[i] = cell;
}
```

Проблема 1: гонка данных

- Тестирование обоих подходов на CUDA GPU с CC 2.0 (Fermi)



Проблема 2: реализация паттерна parallel scan

- Исходный метод:

$$z_P = d_P^{-1}\left(f_P - (A\phi^k)_P - \sum_{S<P} L_{SP} z_S\right)$$

$$x_P = z_P - d_P^{-1} \sum_{S>P} U_{SP} x_S; \quad P = \overline{1, N}$$

$$\phi_P^{k+1} = \phi_P^{k+1} + x_P$$

- Схематичная реализация первого шага

```
for (int p = 0; p < cells.size(); p++) {
    real val = g(p);
    for (int i = 0; i < neighbours[p].size(); i++) {
    {
        int s = neighbours[p][i];
        if (p > s)
            val += out[p] * coeff[s];
    }
    out[p] = val;
}
```

Невозможно сделать параллельную реализацию для общего случая

Проблема 2: реализация паттерна parallel scan

- Решение: переход к итерационному варианту

```
for (int iter=0; iter < 10; iter++) {
    for (int p = 0; p < cells.size(); p++) {
        real val = g(p);
        for (int i = 0; i < neighbours[p].size(); i++) {
        {
            int s = neighbours[p][i];
            if (p > s)
                val += out1[p] * coeff[s];
        }
        out2[p] = val;
    }
    swap(out1, out2);
}
```

- Следствие 1: требуется в разы больше операций
- Следствие 2: возможно создать реализацию для GPU

Проблема 3: нехватка регистров

- Пример заголовка одного требуемого CUDA-ядра:

```
__global__ void fvmDivcd_Faces(
        real *cdif, real *spk, real *ap,
        real *rhs_x, real *rhs_y, real *rhs_z,
        real *face_aps, real *face_asp,
        int *faces_idp, int *faces_ids, real *faces_area, real *faces_g1,
        real *faces_norm_x, real *faces_norm_y, real *faces_norm_z,
        real *faces_fac,
        real *rcv_x, real *rcv_y, real *rcv_z,
        real *faceflux,
        real *fi_scalar_0, real *fi_scalar_1, real *fi_scalar_2,
        Vec3<real *> fi_scalar_0_grad,
        Vec3<real *> fi_scalar_1_grad,
        Vec3<real *> fi_scalar_2_grad,
        real _tiny, int nfaces);
```

34 аргумента!

- Проблема: скомпилированное ядро «вылетает» с ошибкой
- Проблема: на CC 1.3 не хватает памяти для аргументов

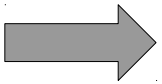
Схема выступления

- О пакете SigmaFlow
- Этап 1. Подготовка
- Этап 2. Портирование
- **Этап 3. Оптимизация**
- Результаты

Оптимизация работы с памятью

- Переиспользование временных буферов на GPU

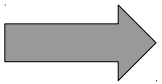
```
void Kernel()
{
    int *buf;
    cudaMalloc(&buf, size);
    //...
    cudaFree(buf);
}
```



```
void Kernel()
{
    int *buf = GetBuffer(size);
    cudaMalloc(&buf, size);
    //...
    ReleaseBuffer(buf);
}
```

- Отказ от синхронного обнуления массивов

```
class Array
{
    int *gpuMem, *cpuMem;
    void Clear()
    { cudaMemset(gpuMem, ...);
      memset(cpuMem, ...); }
};
```

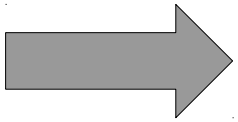


```
class Array
{
    int *gpuMem, *cpuMem;
    void Clear()
    { if (onGpu)
        cudaMemset(gpuMem, ...);
      else
        memset(cpuMem, ...); }
};
```

Изменение вывода в log-файлы

- Замена std::cout на printf()
- Отказ от операции flush на каждой итерации

```
while (eps > 10e-6)
{
   //...
   std::cout << «Iteration #» << N;
   std::cout.flush()
}
```

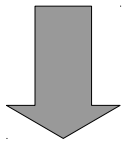


```
while (eps > 10e-6)
{
   //...
   printf(«Iteration #%d», N);
}
```

Оптимизация ядер с помощью TTG Apptimizer

- Шаг 1. Для основных CUDA-ядер вводится дополнительный параметр тяжеловестности нитей

```
__global__ void Copy(int *dst, int *src)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x
    dst[i] = src[i];
}
```



```
template <int N>
__global__ void Copy(int *dst, int *src)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x * N;
    int buf[N];
#pragma unroll
    for (int j = 0; j < N; j++)
        buf[i + j * blockDix.x] = src[i + j * blockDix.x];
#pragma unroll
    for (int j = 0; j < N; j++)
        dst[i + j * blockDix.x] = buf[i + j * blockDix.x];
}
```

Оптимизация ядер с помощью TTG Apptimizer

- Шаг 2. Подбор данного параметра, а также размера блока, поручается автотюнеру.

```
EnumParameter<int> blockWidth(«{64, 128, 256, 512}»);
EnumParameter<int> pointsPerThread(«{ 1, 2, 4, 8}»);

//...
dim3 threads(blockWidth);
dim3 grid(size / blockWidth);
CopyN<<<grid, threads>>>(dst, src, pointsPerThread);
```

В зависимости от типа / размера сеток и модели GPU значения для параметров `blockWidth` и `pointsPerThread` будут подменяться на оптимальные

Результаты

- Эффект от проведённых оптимизаций
  - Переиспользование временных буферов: +23%
  - Отказ от излишних вызовов memset(): +15%
  - Переход на printf(): +5-7%
  - Использование TTG Apptimizer: +10-15%
  - Оптимизация отдельных ядер: +30%
- Итоговое ускорение:

  x2.1 раза, или 6.5 минут против 14 на сетке из 1 млн. узлов.

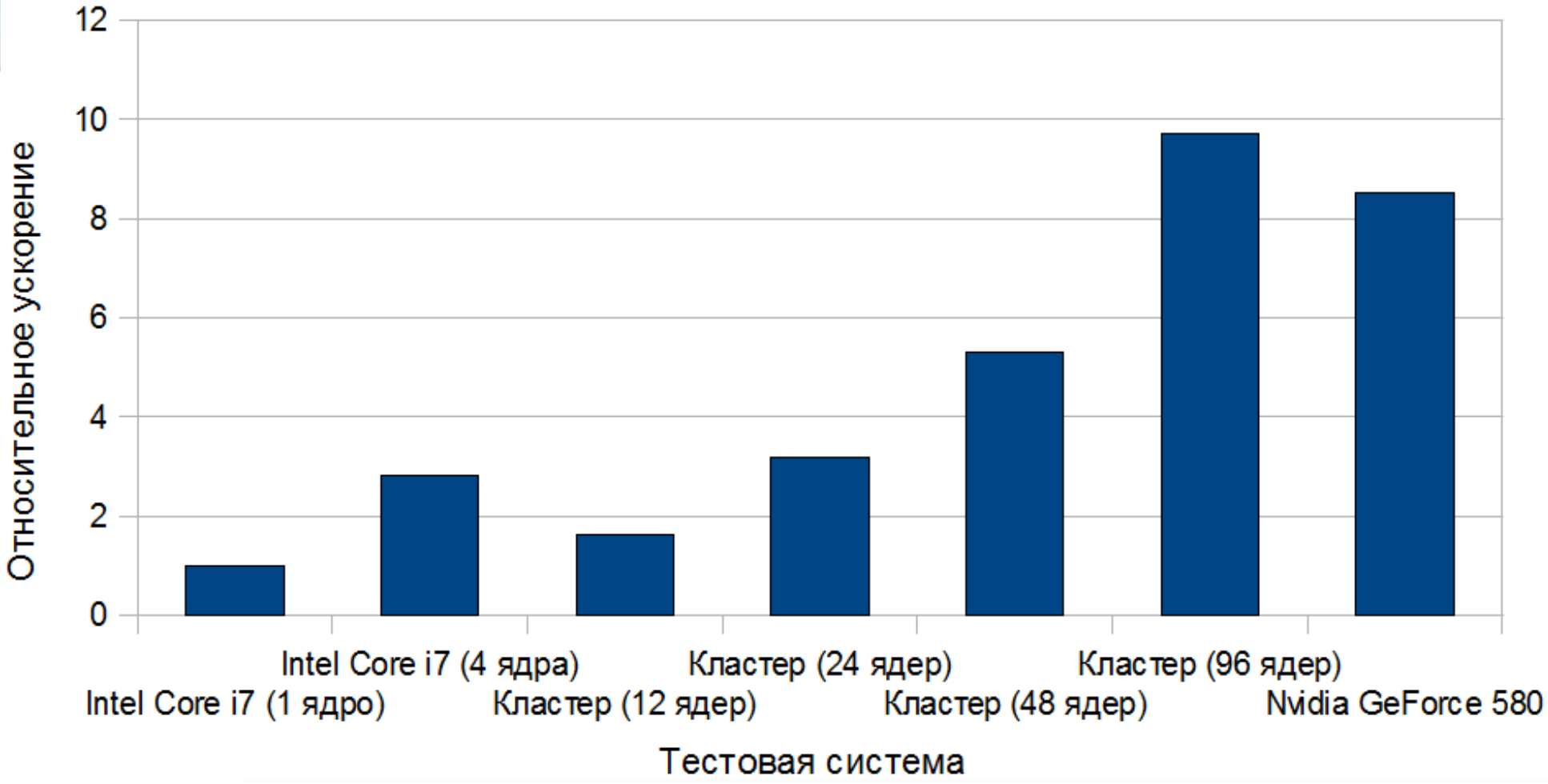
Схема выступления

- О пакете SigmaFlow
- Этап 1. Подготовка
- Этап 2. Портирование
- Этап 3. Оптимизация
- **Результаты**

Тестовые системы

- GPU NVidia GeForce 580 GTX
  - 512 ядер @ 1.5 GHz
  - Порядка 1.5 GFlops на одинарной точности
- CPU Intel Xeon E3-12x0 / Intel Core i7
  - 4 ядра @ 3.4 / 4.4 GHz
  - 108 / 140 GFlops на одинарной точности
- Кластер из узлов IBM Blade HS21
  - 12 узлов, суммарно 96 ядер @ 2.33 GHz
  - 895 GFlops на одинарной точности

Результаты тестирвоания на сетке из 4 млн. узлов



Результаты тестирования на разных сетках