

# Технология адаптации CUDA программ для гибридных (ГПУ/ЦПУ/Xeon Phi) вычислений

А.В. Кашковский, А.А. Шершнёв, П.В. Ващенко



Новосибирский государственный университет



Институт теоретической и прикладной механики  
им. С.А. Христиановича СО РАН

630090 Новосибирск, Россия.  
e-mail: {sasa,antony,vashen}@itam.nsc.ru

01.02.2018

- Для больших задач нужны большие вычислительные мощности
- ГПУ+CUDA – один из эффективных способов решения
- НО!
  - В России трудно найти кластер с достаточным числом ГПУ
  - Появляются мощные, многоядерные ЦПУ
  - Развивается Xeon Phi
  - Современные вычислительные суперкомпьютеры имеют гибридную архитектуру (ЦПУ+ГПУ, ЦПУ+Phi)
- Хочется иметь программу, которая могла бы проводить вычисления на любой архитектуре и их комбинации
- Хочется, чтобы исходный текст программы был един для всех платформ
- Одно из решений – OpenCL, но что делать с программами на CUDA?

# Основные принципы адаптации

- Использовать технологию OpenMP для вычислений на многоядерных устройствах
  - все потоки OpenMP и CUDA имеют доступ к общей памяти → параллелизация по данным
  - многоядерный ЦПУ можно представить в виде одного блока CUDA, в котором число CUDA тредов равно числу ЦПУ ядер
  - для каждого ЦПУ потока вызывать аналог CUDA kernel
- Исходный текст должен быть одинаков для всех платформ
- Для модификации исходного текста используется препроцессор языка C/C++
- Желательно уменьшить число директив препроцессора
- Часть функций может быть переписана индивидуально под каждую платформу, но таких функций должно быть как можно меньше
- Гетерогенные вычисления делаются с помощью MPI

# Пример программы на CUDA и OpenMP

## CUDA:

```
--global__ void SumDx(  
    double* x,    // array of data [len]  
    double* dx,  // array of increments [len]  
    double t,    // time step  
    int len      // length of arrays x and dx  
)  
{  
    register int tid = blockDim.x * blockIdx.x + threadIdx.x; // global index of thread  
    register int nth = blockDim.x * gridDim.x;                // total number of threads  
  
    for (;tid<len; tid+=nth) {  
        x[tid] += dx[tid]*t;  
    }  
}
```

```
int blocks = 32;  
int threads = 256;  
SumDx<<<blocks, threads>>>(x, dx, t, len);
```

## OpenMP:

```
#pragma omp parallel for  
for (tid=0; tid<len; tid++) {  
    x[tid] += dx[tid]*t;  
}
```

# Конвертирование CUDA kernel для OpenMP

На OpenMP всегда будет один block,  
а число threads равно числу OpenMP потоков.

```
static _uint3 blockDim;           // Number of threads variable
static _uint3 gridDim(1);        // Number of blocks = 1
static _uint3 blockIdx(0);       // Index of block = 0
```

```
blockDim.x = omp_get_max_threads(); // Number of threads = Number of OMP threads
omp_set_dynamic(0);                 // turn off dynamic Number of OMP threads
omp_set_num_threads(blockDim.x);    // set static Number of OMP threads
```

```
#pragma omp parallel for
for (unsigned int omp=0; omp<blockDim.x; omp++) {
    _uint3 threadIdx(omp,0,0);
    SumDx_OMP(threadIdx, x, dx, t, len);
}
```

# OpenMP “kernel”

```
void SumDx_OMP(  
    _uint3 threadIdx, // !!! index of OMP thread  
    double* x, // array of data [len]  
    double* dx, // array of increments [len] [len]  
    double t, // timestep  
    int len // length of arrays x and dx  
)  
{  
    register int tid = blockDim.x * blockIdx.x + threadIdx.x; // global index of thread  
    register int nth = blockDim.x * gridDim.x; // total number of threads  
    for ( ; tid < len; tid += nth) {  
        x[tid] += dx[tid]*t;  
    }  
}
```

- Отличие “kernel” только в дополнительном параметре функции.
- Делается макросами препроцессора `KERNEL_FUNC` и `KERNEL_CALL`.

Исходный текст:

```
#include "kernel_wrap.h"  
...  
// declaration of a function  
KERNEL_FUNC(SumDx, double* x, double* dx, double t, int len);  
...  
// call of a function  
KERNEL_CALL(SumDx, blocks, threads, x, dx, t, len);
```

# kernel\_wrap.h файл

```
#ifdef KERNELWRAP
// OMP part
...
// declaration of a function
#define KERNEL_FUNC( NNN, ... ) void NNN##_OMP ( _uint3 threadIdx, __VA_ARGS__ )
// call of a function
#define KERNEL_CALL( NNN, BLK, TID, ... ) \
_Pragma ("omp parallel for") \
    for (unsigned int omp=0; omp<blockDim.x; omp++) {\
        _uint3 threadIdx(omp,0,0); \
        NNN##_OMP ( threadIdx, __VA_ARGS__ ); \
    }
...
#else
// CUDA part
...
// declaration of a function
#define KERNEL_FUNC( NNN, ... ) __global__ void NNN ( __VA_ARGS__ )
// call of a function
#define KERNEL_CALL( NNN, BLK, TID, ... ) NNN<<<BLK,TID>>>( __VA_ARGS__ )
...
#endif
```

# Подмена CUDA функций

В OpenMP разделе файла `kernel_wrap.h` также надо подменить специфичные для CUDA определения и функции:

```
#define __global__
#define __host__
#define __device__
#define __shared__ static

inline void cudaMalloc(void** ptr, size_t size) {
    (*ptr) = (void *)new char[size];
}

template <class T> inline void cudaFree(T * ptr) { delete [] (char*)ptr; }

inline void* cudaMemcpy( void* dest, void*
    return (memcpy(dest, src, size));
}

template <class T> inline T atomicAdd(T* address, T val)
{ T rc;
#pragma omp atomic capture
    { rc=(*address);
      (*address)+=val;
    }
    return(rc);
}
```

# Платформа-зависимые функции и сборка

## Вставка платформа-зависимых функций

```
#ifdef KERNELWRAP
// OpenMP version
MyFunction();
#else
// CUDA version
cuMyFunction<<<block, thread>>>();
#endif
```

## Сборка для CUDA

```
$ nvcc -c sum.cu -o sum.o
...
$ nvcc main.o sum.o -o main_cuda
```

## Сборка для OpenMP

```
$ g++ -DKERNELWRAP -c sum.cu -o sum.obj
...
$ g++ main.obj sum.obj kernel_wrap.o -o main_omp
```

# Апробация

- Программа HyCFS: решение уравнений Навье-Стокса на регулярной сетке, WENO схема 5-го порядка (80% счета)
- 13 тыс. строк исходного текста
- 327 строк изменено ( $\sim 2.5\%$ )

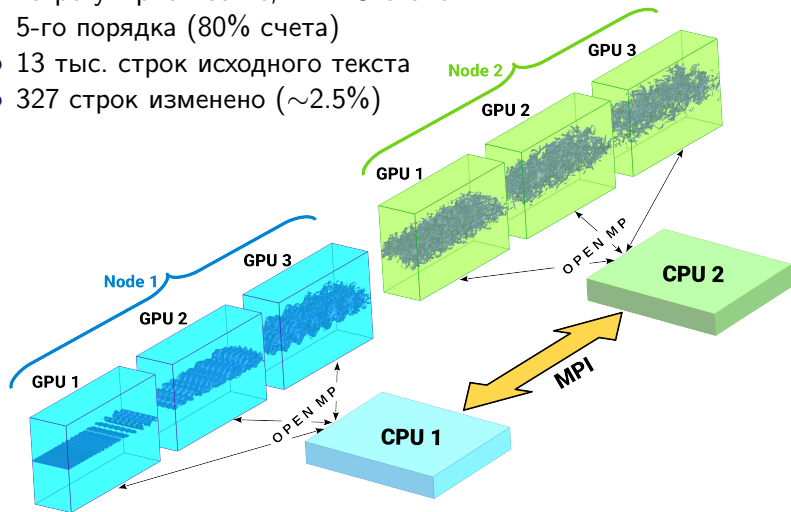
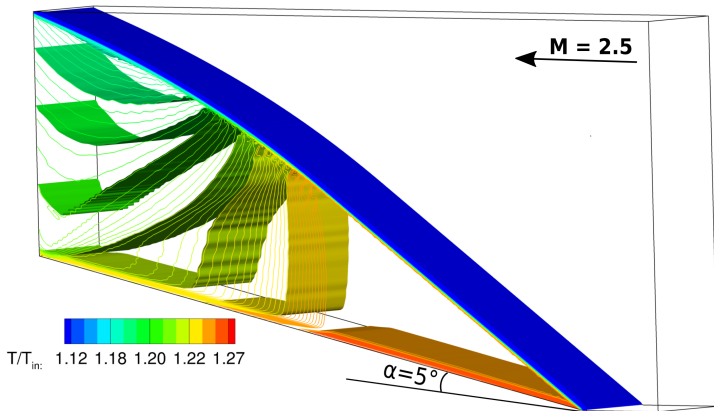


Схема параллелизации HyCFS

# Тестовый пример

Сверхзвуковое обтекание клина,  $M=2.5$

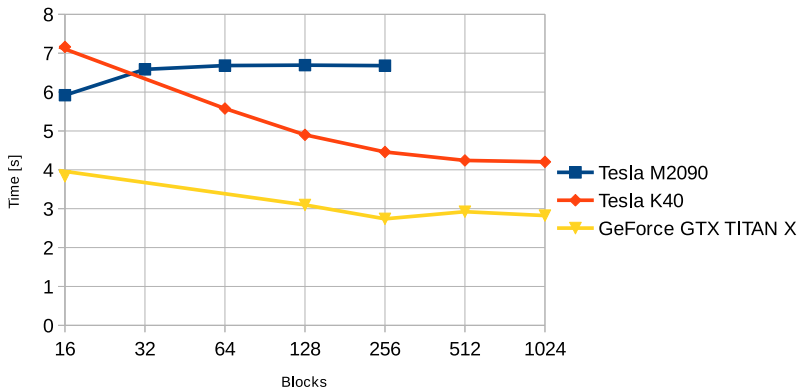
Сетка  $200 \times 100 \times 420 = 8.4$  млн. ячеек



Изоповерхности температуры

# Расчеты на ГПУ

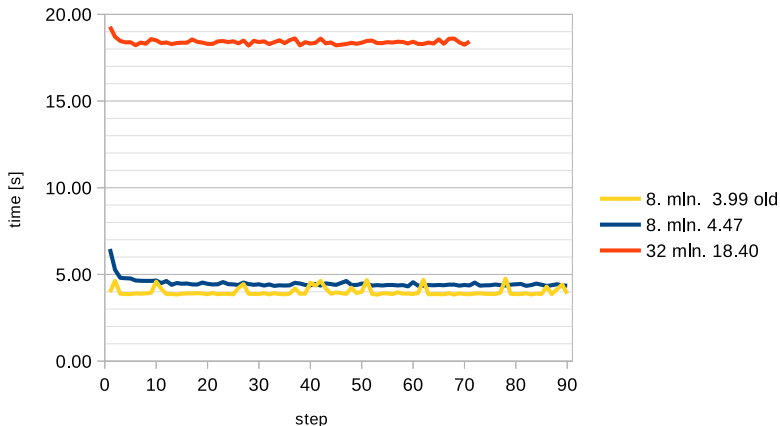
Среднее время вычисления одного шага. Число нитей : 256



!!! Время вычисления зависит от числа блоков.

# Расчеты на Xeon Phi (KNL)

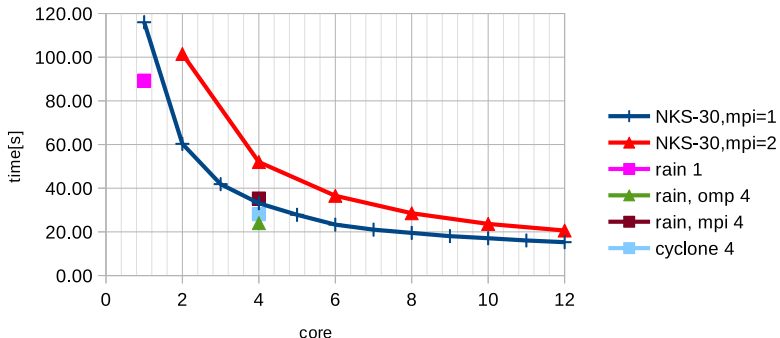
- Число нитей : 288
- Две сетки:  $\sim 8$  и  $\sim 32$  млн. ячеек
- До модернизации кластера (old) наблюдаются всплески загрузки



Время выполнения каждого шага

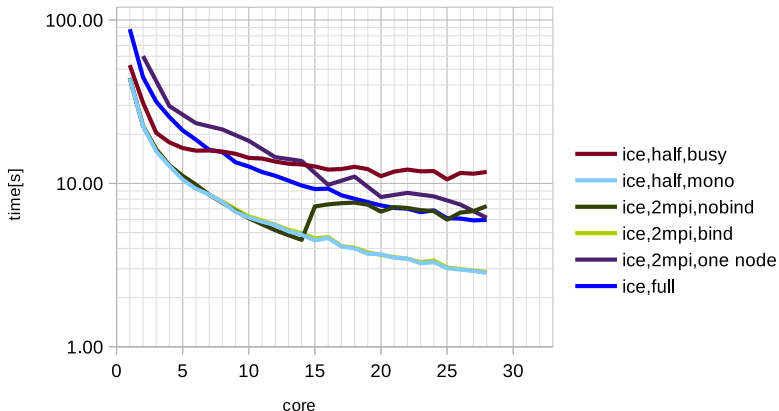
# Расчеты на ЦПУ

| Название | Модель             | Частота | Ядер |
|----------|--------------------|---------|------|
| НКС-30   | Intel Xeon X5675   | 3.07GHz | 12   |
| rain     | Intel Core i5-4430 | 3.00GHz | 4    |
| cyclone  | Intel Core i5-4460 | 3.20GHz | 4    |



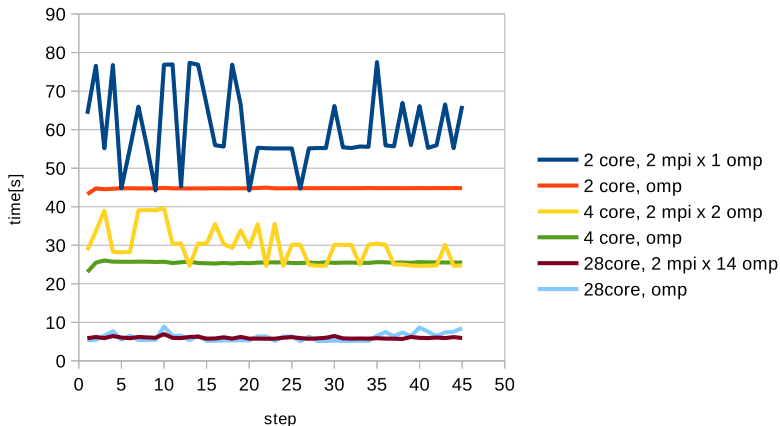
# Расчеты на ЦПУ, iceberg

- 2 node  $\times$  2 Intel Xeon E5-2683 v3 @ 2.00GHz  $\times$  14 core
- Часть расчетов делалась для гибридизации (две подобласти)
- Важна опция компиляции `-bind-to none`
- 2 MPI на одном узле мешают друг другу.



# OMP vs MPI

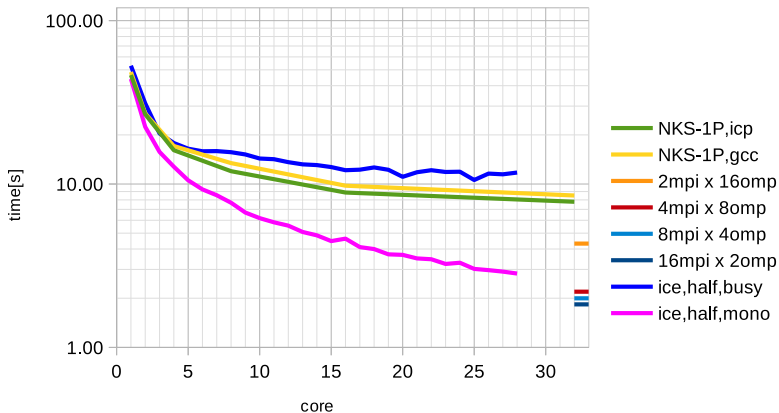
- iceberg: 2 node  $\times$  2 Intel Xeon E5-2683 v3 @ 2.00GHz  $\times$  14 core
- MPI на одном узле + малое число OMP – большие задержки



Время выполнения каждого шага

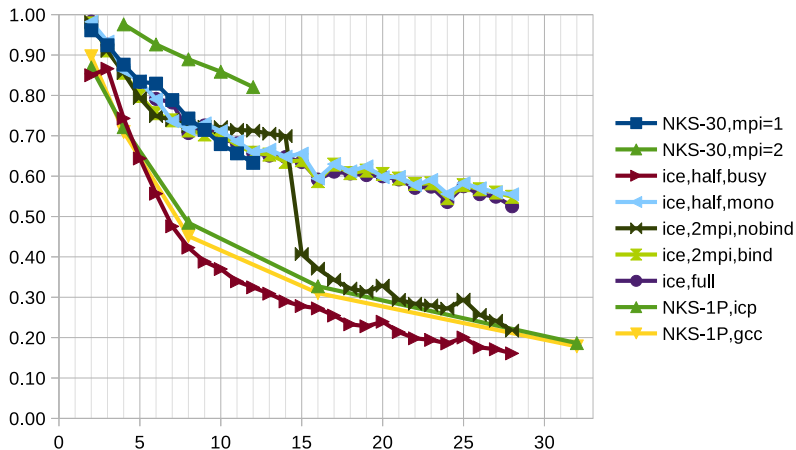
# Расчеты на ЦПУ, НКС-1П (Broadwell)

- Сильное замедление с увеличением OMP потоков
- Нужна помощь для анализа загрузки



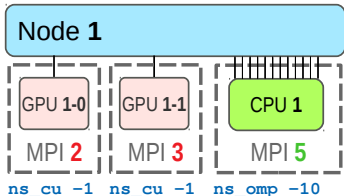
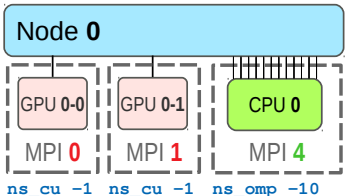
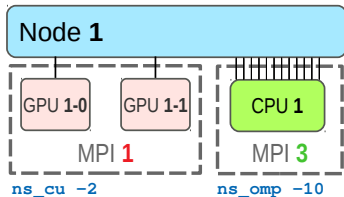
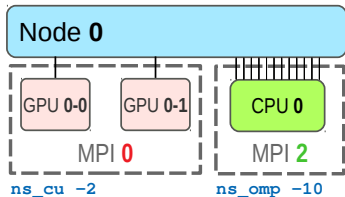
# Эффективность параллелизации на ЦПУ

$$E_p = \frac{T_N}{T_1 \cdot N}$$



# Гибридные вычисления

- Две исполняемых программы (ns\_cu – GPU, ns\_omp – CPU).
- Запуск и обмен данными с помощью MPI
- ВАЖНО правильно запустить программы на узлах. mpirun – проще запускать программы одного типа, потом другого.



# Запуск гибридных вычислений

Фрагмент qsub.sh

2-х узла, на каждом 1 GPU и CPU  $\times$  27 ядер:

```
#PBS -l nodes=2:ppn=2

# GPU nodes
cat $PBS_NODEFILE | sort | uniq > myhosts
# CPU nodes (the same!)
cat $PBS_NODEFILE | sort | uniq >> myhosts

mpirun --bind-to none --map-by node --hostfile myhosts \
  -np 2 /home/sasa/Progs/NS_CUDA/Bin/ns_cu -1 : \
  -np 2 /home/sasa/Progs/NS_CUDA/Bin/ns_omp -27
```

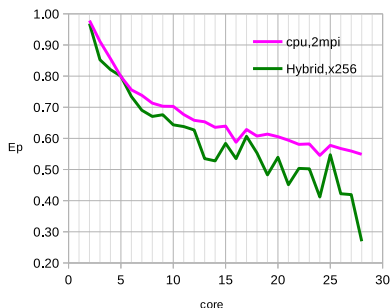
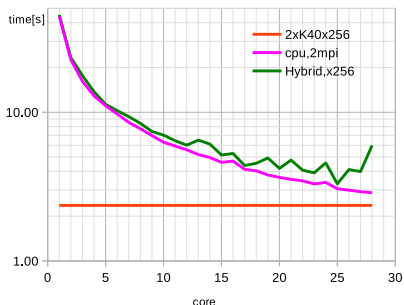
# Гибридные вычисления на уровне программы

- Для каждого устройства необходимо корректно найти назначенную ему подобласть. Важна “топология” задания.
- Рекомендуется, для каждого MPI процесса печатать:
  - Имя запускаемого файла
  - Имя узла
  - MPI rank
  - Индекс обслуживаемых устройств (GPU)
  - Индекс подобласти
- Нужен баланс загрузки процессоров. **Но!** Из-за флуктуаций не гарантируется равномерность загрузки.
  - Динамический – переконфигурация области в процессе вычислений. **Универсально** . **Сложно**
  - Статический – конфигурация области до вычислений. Необходимо поделить область пропорционально производительности GPU и CPU. **Минимальные изменения в программе** . **Нужно знать соотношение производительности**

# Результаты гибридного расчета

- Iceberg: Времена счета GPU и OMP близки. Без баланса загрузки.

| Узлов | GPU на узел | OMP на узел | лучшее время на шаг |
|-------|-------------|-------------|---------------------|
| 1     | 2           | 0           | 2.36                |
| 2     | 0           | 27          | 2.87                |
| 1     | 1           | 27          | 3.99                |
| 2     | 1           | 27          | 1.55                |



- Показано, что CUDA программы можно конвертировать для OpenMP вычислений.
- Удалось достичь “архитектурной гибкости” программ
- Для выполнения гибридных вычислений нужны минимальные переделки.
- Эффективность вычислений на CPU, как правило, выше 50%
- Появилась возможность отладки и профилирования программ на CPU
- Нужно привлекать ПО профилирования для оптимизации программ.

# kernel\_wrap.h

```

/*****
 * wrapper for converting CUDA function OpenMP
 * 14.06.2016 A.V.kashkovsky
 *****/

#ifndef KERNELWRAP_H
#define KERNELWRAP_H

// Wrapper for CUDA/OpenMP
#ifdef KERNELWRAP

#include <string.h>

// Redefine classes

class _uint3 {
public:
    unsigned int x, y, z;
    _uint3(unsigned int X = 0,
            unsigned int Y = 0,
            unsigned int Z = 0): x(X), y(Y), z(Z) {}
};

typedef _uint3 uint3;

class dim3 {
public:
    unsigned int x, y, z;
    #if defined(__cplusplus)
        dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
        dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
        operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
    #endif /* __cplusplus */
};

typedef int cudaError_t;
typedef double * cudaEvent_t;

enum cudaMemcpyType {
    cudaMemcpyDeviceToHost = 1,
    cudaMemcpyHostToDevice,
    cudaMemcpyHostToHost,
    cudaMemcpyDeviceToDevice
};

static int cudaSuccess = 1;

// global elements
extern uint3 blockIdx;
extern uint3 blockDim;
extern uint3 gridDim;

// part of OpenMP declarations
#define __global__
#define __host__
#define __device__
#define __shared__ static

#define KERNEL_FUNC( NNN, ... ) void NNN##_OMP ( _uint3 threadIdx, __VA_ARGS__ )
#define KERNEL_DECLARE( NNN, ... ) void NNN##_OMP ( _uint3 TID, __VA_ARGS__ )

#define KERNEL_CALL( NNN, BLK, TID, ... ) \
#pragma ("omp parallel for") \
for( unsigned int omp=0; omp<blockDim.x; omp++) \
{ _uint3 blockIdx(omp,0,0); \
  NNN##_OMP ( blockIdx, __VA_ARGS__ ); \
}

inline void __syncthreads () {
    #pragma omp barrier
}


```

```

inline void cudaThreadSynchronize(){}
inline void cudaDeviceSynchronize(){}

template <class T> inline T atomicAdd(T* address, T val)
{ T rc;
#pragma omp atomic capture
{ rc=(*address);
  (*address)+=val;
}
return(rc);
}

// ---- declaration of functions
void kernelwrap_Init(int num_thread=0);

inline void cudaMalloc(void** ptr, size_t size) {
  (*ptr) = (void *)new char[size];
}

template <class T> inline void cudaFree(T * ptr) { delete [] ptr; }

inline void* cudaMemcpy(
    void* dest,
    void* src,
    size_t size,
    cudaMemcpyType tt
) {
  return (memcpy(dest, src, size));
}

inline void cudaMemset(void* pt, int z, size_t size) { memset(pt, z, size); }

inline int cudaGetLastError(){return 0;}

inline char* cudaGetErrorString(int code) {
  static char str[] = "unknown";
  return str;
}

inline int cudaSetDevice(int d) {return((d==0)?cudaSuccess:0);}
inline int cudaGetDevice( int *d ){*d=0; return(cudaSuccess);}

void cudaEventCreate(cudaEvent_t *d );
void cudaEventRecord(cudaEvent_t d, int i );
void cudaEventSynchronize(cudaEvent_t d );
void cudaEventElapsedTime(float *d, cudaEvent_t a, cudaEvent_t b );
void cudaEventDestroy(cudaEvent_t d );

#else

#define KERNEL_FUNC( NNN, ... ) __global__ void NNN ( __VA_ARGS__ )

#define KERNEL_CALL( NNN, BLK, TID, ... ) NNN<<<BLK,TID>>>(< __VA_ARGS__ )

#endif

#define OMP_CALL( NNN, BLK, TID, ... ) NNN##_OMP ( (TID), (BLK), __VA_ARGS__ )

#define CU_CALL( NNN, BLK, TID, ... ) NNN<<<BLK,TID>>>(< __VA_ARGS__ )

#endif

```

# kernel\_wrap.cc

```

/*****
 * wrapper to converting CUDA to OpenMP
 * 01.03.2017 Kashkovsky A.V.
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#include <omp.h>

#define KERNELWRAP
#include "kernel_wrap.h"

// some global elements
uint3 blockDim(1);
uint3 gridDim(1);
uint3 blockIdx;

/*****
 * kernelwrap_Init
 * initialisation of OpenMP parameters
 *
 * 02.03.2017 Kashkovsky A.V.
 *****/
void kernelwrap_Init(int num_threads) {
    if (num_threads == 0) {
        blockDim.x = omp_get_max_threads();
    } else {
        blockDim.x = num_threads;
    }

    omp_set_nested(1); // allow nested parralle
    omp_set_dynamic(0); // deny dynamic change number of threads
    omp_set_num_threads(blockDim.x);
    printf("Number of OpenMP threads: %d\n", blockDim.x );
}

/*****
 *
 * Some standard timer in C
 * 28.11.2013 A.V. Kashkovsky
 *
 *****/
double mytimer() {
    struct timeval tv;
    struct timezone tz;
    gettimeofday(&tv, &tz);
    double t = (double)tv.tv_sec+((double)tv.tv_usec)/1000000.;

    return t;
}

double atimer();
void cudaEventCreate(cudaEvent_t *d ){ (*d)=new double;}
void cudaEventRecord(cudaEvent_t d, int i ){ (*d) = atimer();}
void cudaEventSynchronize(cudaEvent_t d ){ (*d) = atimer();}
void cudaEventElapsedTime(float *d, cudaEvent_t a, cudaEvent_t b ){ (*d)=(*b)-(*a);}
void cudaEventDestroy(cudaEvent_t d ){delete d;}
```

Спасибо за Внимание!