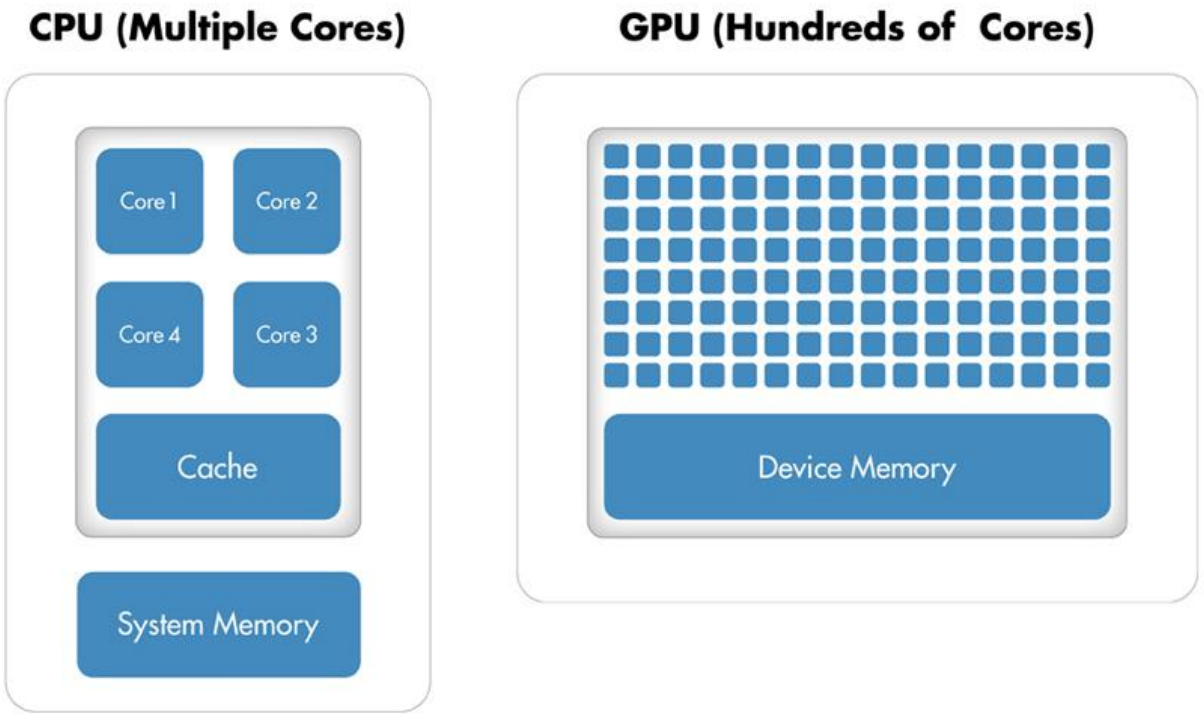
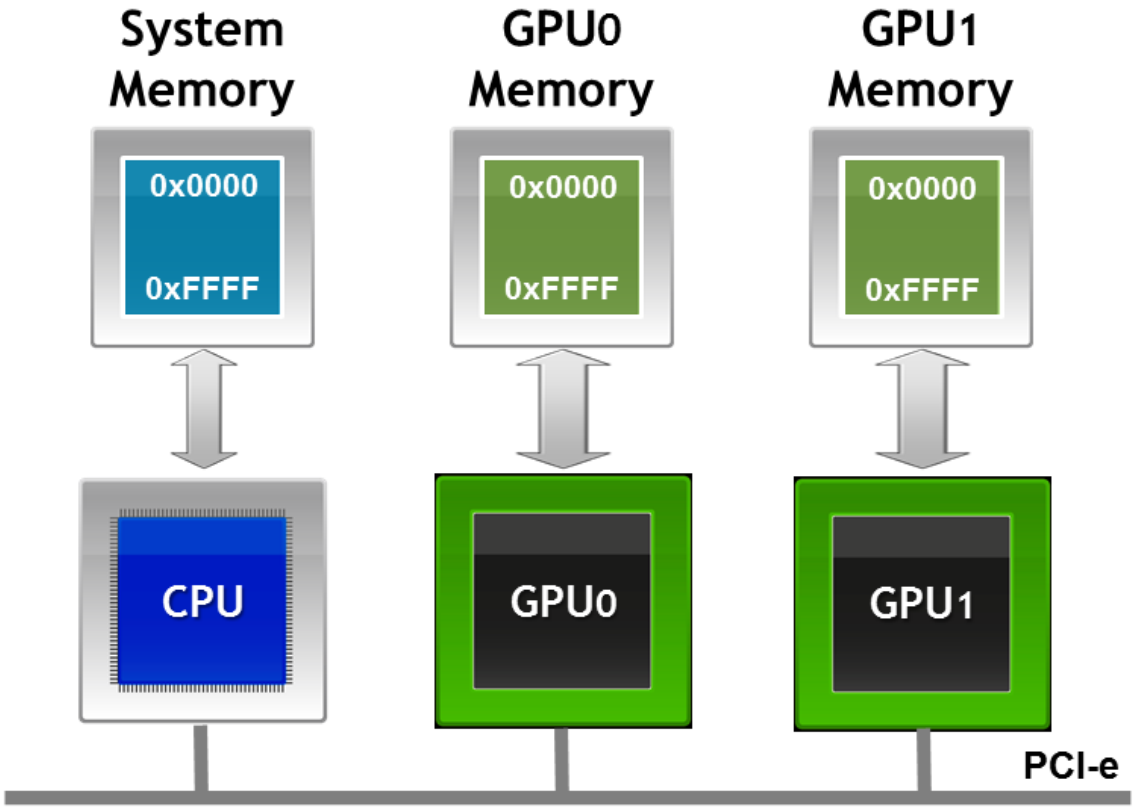
Совместное использование технологий MPI и CUDA

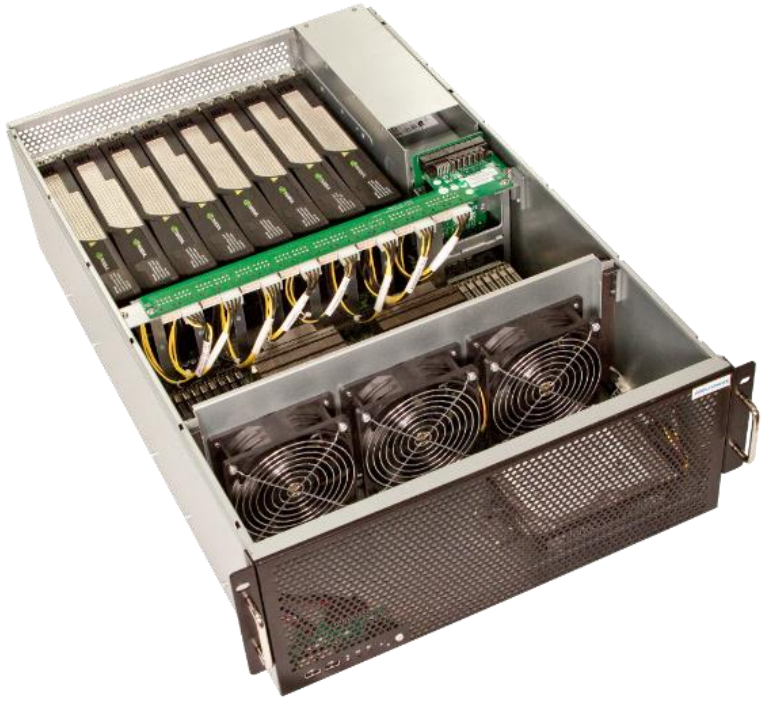
Гущин В. В.
guschin108@gmail.com

Сравнение CPU и GPU

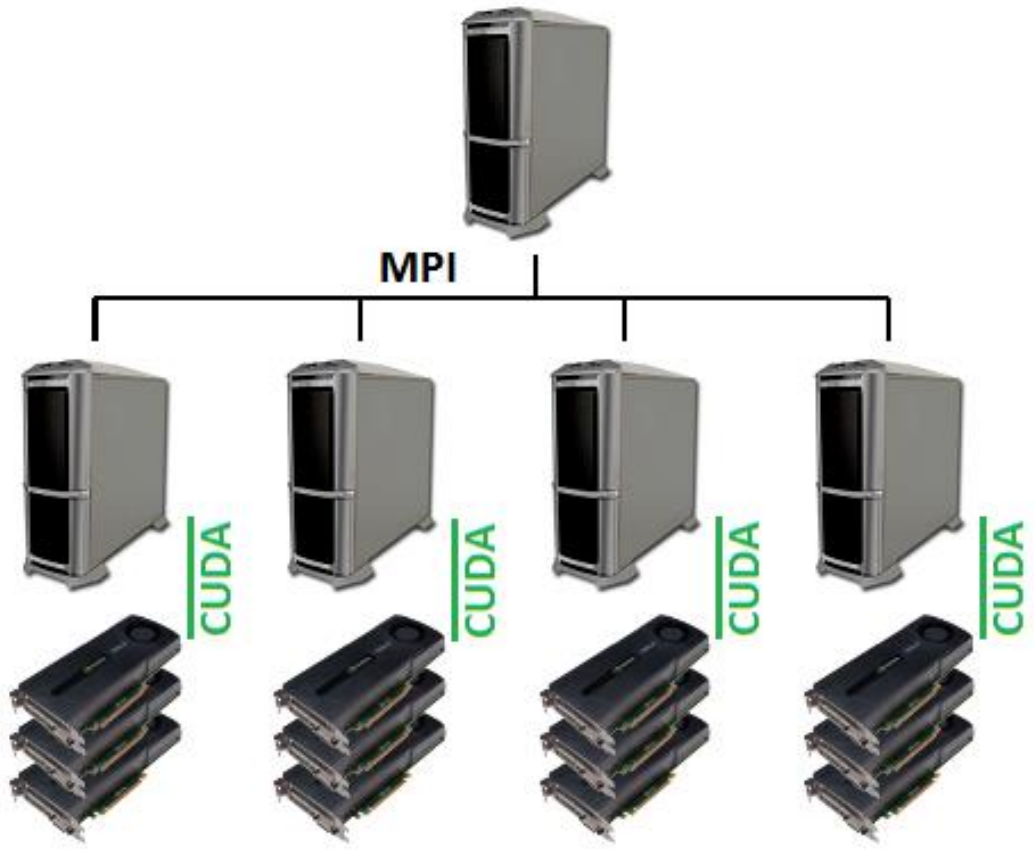


Гибридный вычислительный узел





Гибридная вычислительная система



Логика работы гибридной параллельной программы

- Декомпозиция задачи на управляющем узле;
- Обмен данными между CPU по MPI;
- Загрузка данных с CPU на GPU;
- Загрузка и выполнение ядер на GPU;
- Выгрузка данных с GPU на CPU;
- Обмен данными между CPU по MPI;

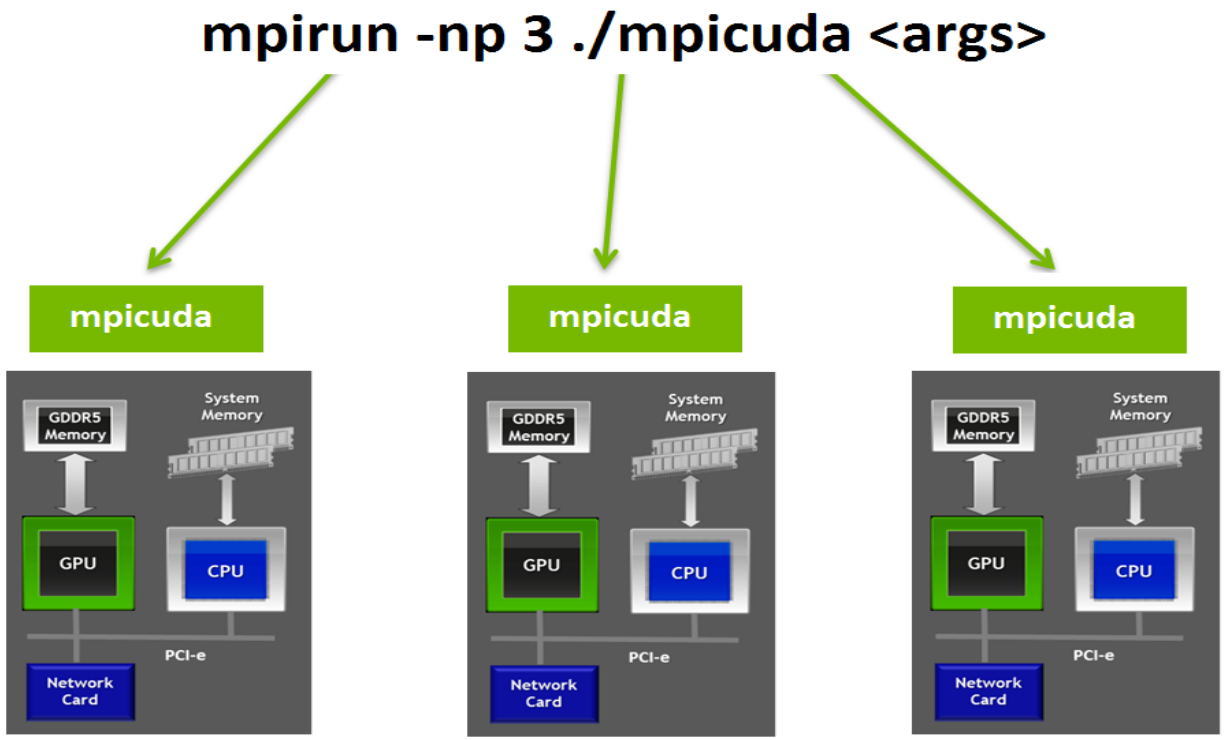
Компиляция гибридной параллельной программы

Для компиляции последовательного и параллельного кода CPU, а также сборки конечной программы используется компилятор **mpicc**. Для компиляции параллельного GPU кода используется компилятор **nvcc**.

Рассмотрим пример. Имеется файл gpu.cu в котором содержится GPU и CPU код (CUDA) и файл main.c в котором содержится CPU код (MPI). Чтобы скомпилировать программу, которая будет использовать технологии CUDA и MPI, нужно совершить несколько последовательный действий.

1) Компиляция кода CUDA в объектный файл
nvcc -c gpu.cu -o gpu.o -L/opt/cuda-5.5/lib64 –lcudart
2) Компиляция кода MPI в объектный файл
mpicc -c main.c -o main.o -L/opt/cuda-5.5/lib64 –lcudart
3) Сборка бинарного файла
mpicc -o mpicuda main.o gpu.o -L/opt/cuda-5.5/lib64 -lcudart -lstdc++

Запуск гибридной параллельной программы



Запуск гибридной параллельной программы аналогичен запуску обычный MPI программы. Для запуска программы mpicuda следует выполнить команду mpirun. Она заботится о запуске нескольких экземпляров программы и распространяет эти экземпляры по узлам.

Совместное использование MPI и CUDA

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
   int size = 0; int rank = 0;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);


   if (0 == rank) // Управляющий узел
       main_node(size);
   else // Обычный вычислительный узел
       second_node(rank);

   return 0;
}
```

Совместное использование MPI и CUDA

```
/* Структура описывающая устройство в системе */
typedef struct
{
   int node_id;
   struct cudaDeviceProp dev_prop;
} dev_info_t;

/* Выделение памяти под массив структур описывающих устройства */
dev_info_t * alloc_buffer_for_devices_info(int cnt)
{
    dev_info_t * dev_info = NULL;
    dev_info = (dev_info_t *)malloc(sizeof(dev_info_t) * cnt);
    if (NULL == dev_info)
    {
        perror("Can not alloc memory");
        exit(1);
    }

    return dev_info;
}
```

Совместное использование MPI и CUDA

```
typdef struct
{
    int node_id;
    int dev_cnt;
} node_t;

/* Получение информации о количестве устройств в системе */
void get_devices_count(node_t * nodes, int * dev_cnt, int size)
{
    MPI_Status status;
    *dev_cnt = 0;
    int it = 0;
    for(it = 1; it < size; ++it)
    {
        nodes[it - 1]->node_id = it;
        MPI_Recv(&nodes[it - 1]->dev_cnt, 1 , MPI_INT, it, 1, MPI_COMM_WORLD, &status);
        *dev_cnt += nodes[it - 1]->dev_cnt;
    }
}
```

Совместное использование MPI и CUDA

```
/* Получение информации о всех устройствах системы */
void get_devices_info(node_t * nodes, dev_info_t * dev_info, int size)
{
   MPI_Status status;
   int idx = 0;
   int it = 0;
   for(it = 1; it < size; ++it)
   {
       size_t msg_len = sizeof(dev_info_t) * nodes[it - 1]->dev_cnt;
       MPI_Recv(dev_info[idx] , msg_len, MPI_CHAR, it, 1, MPI_COMM_WORLD, &status);
       idx += nodes[it - 1]->dev_cnt;
   }
}
```

Получение информации о доступных GPU

```
void main_node(int size)
{
   int dev_cnt = 0;

   /* Запрос количества устройств */
   node_t nodes[size - 1]; memset(&nodes, 0, sizeof(nodes);
   get_devices_count(&nodes, &dev_cnt, size);
   /* Выделение памяти */
   dev_info_t * dev_info = alloc_buffer_for_devices_info(dev_cnt);
   /* Запрос информации об устройствах */
   get_devices_info(dev_info, &nodes, size);

   //TODO: Декомпозиция задачи
   //TODO: Рассылка подзадач по узлам
   //TODO: Получение и обработка результатов

   MPI_Barrier(MPI_COMM_WORLD);

   //TODO: Сохранение результатов вычислений
   free(dev_info);
}
```

Совместное использование MPI и CUDA

```
/* Получение подробной информации об устройствах на узле */
dev_info_t * get_local_devices_info(size_t * size, int rank)
{
    int dev_cnt = 0;
    cudaGetDeviceCount(&dev_cnt);

    *size = sizeof(dev_info_t) * dev_cnt;

    dev_info_t * dev_info = alloc_buffer_for_devices_info(dev_cnt);

    int it = 0;
    for(it = 0; it < dev_cnt ; ++it)
    {
        dev_info[it]->node_id = rank;
        cudaGetDeviceProperties(&dev_info[it]->dev_prop, it);
    }

    return dev_info;
}
```

Совместное использование MPI и CUDA

```
/* Передача подробной информации об устройствах на узле */
void give_devices_info(dev_info_t * dev_info, size_t msg_len)
{
    MPI_Send(dev_info, msg_len, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
}
```

Совместное использование MPI и CUDA

```
void second_node(int rank)
{
   MPI_Status status;
   size_t msg_len = 0;
   /* Запрос доступных устройств на узле */
   dev_info_t * dev_info = get_local_devices_info(&msg_len, rank);
   /* Отправка управляющему узлу информации об доступных устройствах */
   give_devices_info(dev_info, msg_len);

   //TODO: Получение подзадачи
   //TODO: Обработка подзадачи
   //TODO: Передача результатов управлявшему узлу

   MPI_Barrier(MPI_COMM_WORLD);
   free(dev_info);
}
```

Вызов ядра CUDA из «C» программы

kernel.cu

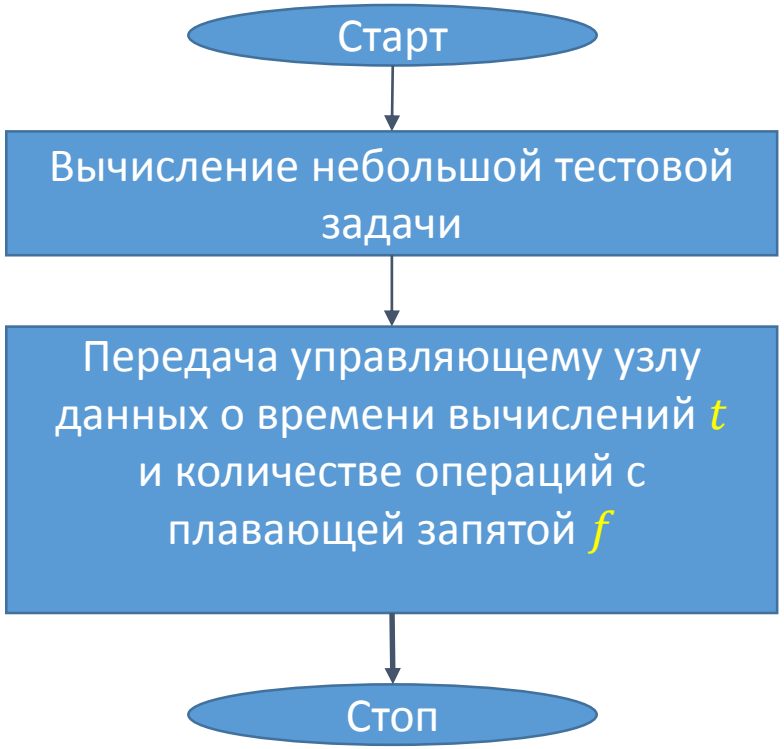
```
#include<stdio.h>
#include<cuda_runtime.h>
#include<cuda.h>

__global__ void mulmatrix(float *a, float *b, float *c, int n)
{

   int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   float sum = 0.0f;

   for(int i  = 0; i < n; i++)
   {
       sum += a[xIndex * n + i] * b[i * n + yIndex];
   }

   c[xIndex * n + yIndex] = sum;

}
```

Вызов ядра CUDA из «C» программы

kernel.cu

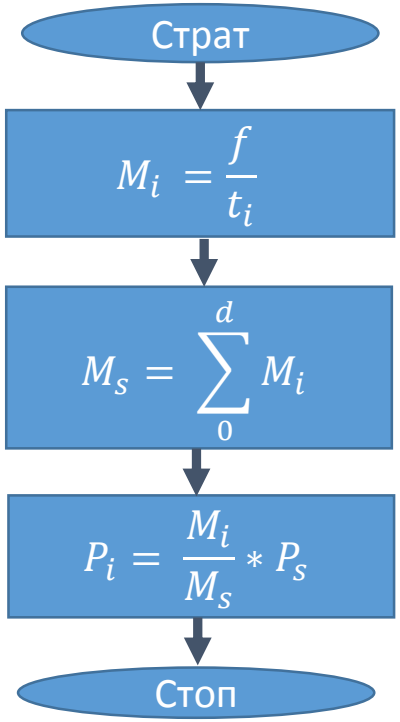
```
extern "C" void gpu_matrix_mul(struct task * t)
{
   int blocks_x;
   dim3 blocks;
   dim3 threads  = dim3(BLOCK_SIZE, BLOCK_SIZE);

   for(int it = 0; it < t->device_count; ++it)
   {
      blocks_x = *(t->size)/sizeof(float)/t->N/BLOCK_SIZE;
      blocks = dim3(t->N/BLOCK_SIZE, t->N/BLOCK_SIZE);

       cudaSetDevice(i);
       mulmatrix<<< blocks, threads >>> (*(t->A_dev + it), *(t->B_dev + it),
              *(t->C_dev + it), t->N);
   }

}

//Метод вычисляет умножение матриц A и B. Результат сохраняется в матрицу C.
```

Балансировка нагрузки между GPU разной мощности



Данный алгоритм выполняется на всех узлах гибридной вычислительной системы.

Балансировка нагрузки между GPU разной мощности



M_i – коэффициент производительности GPU.

M_s – сумма коэффициентов производительности.

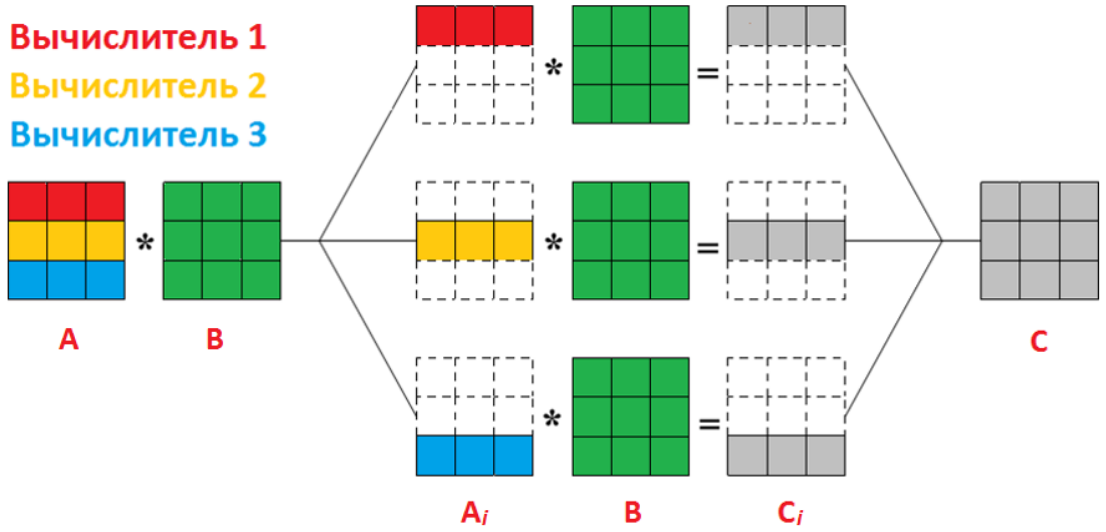
P_s – количество подзадач.

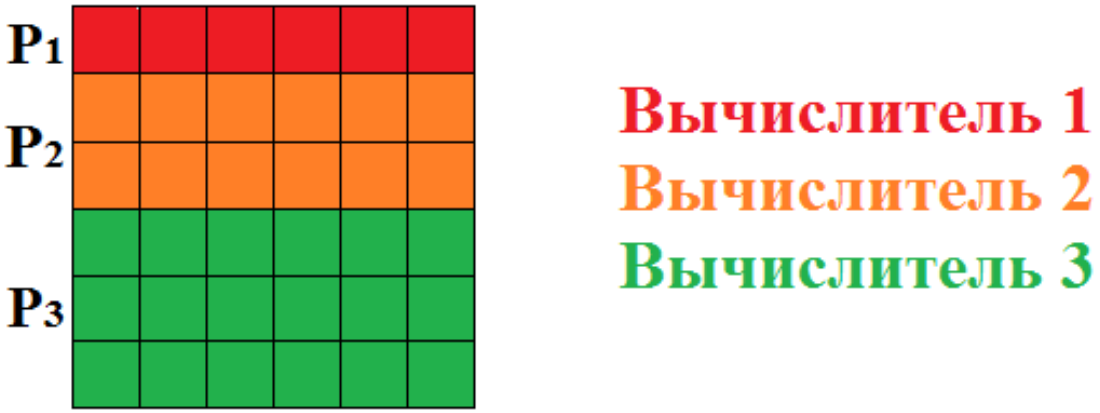
P_i – количество подзадач для GPU.

Данный алгоритм выполняется на управляющем узле гибридной вычислительной системы. В результате выполнения данного алгоритма мощные GPU получат большее количество подзадач.

Балансировка нагрузки между GPU разной мощности

```
[guschin@cngpu1 bin]$ ./all.sh
# Node : 0
device : 0 | name : GeForce GTX 680
# Node : 1
device : 0 | name : GeForce 210
device : 1 | name : GeForce 210
# Node : 2
device : 0 | name : GeForce GT 630

Enter N : 4096

Size task on device
device 0 : 55836672 bytes
device 1 : 1048576 bytes
device 2 : 1048576 bytes
device 3 : 9175040 bytes

Run time 12.141112 sec
```





Заключение

Совместное использование технологий MPI и CUDA позволяет получить значительный прирост производительности, так как появляется возможность объединить большое количество графических процессоров для вычисления одной общей задачи.