# Использование CUDA на Multi-GPU кластере

Дмитрий Бикулов
dmitry@bikulov.org
Физический факультет МГУ
кафедра компьютерных методов физики

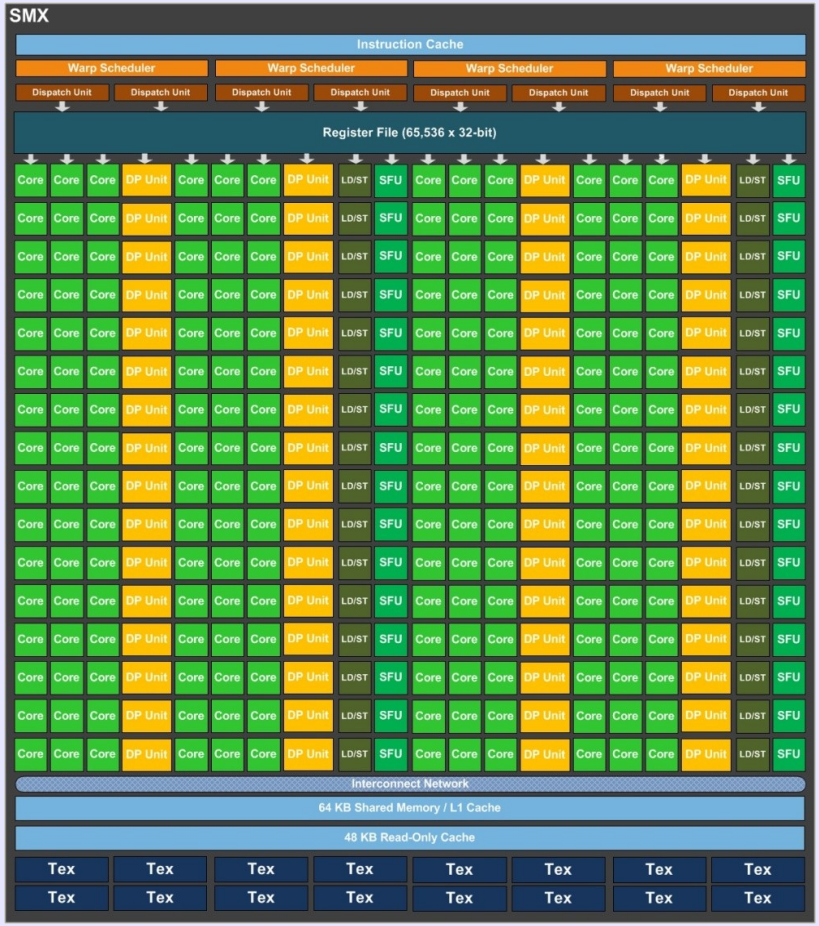
# Архитектура Fermi

# Архитектура Kepler

# Мультипроцессор

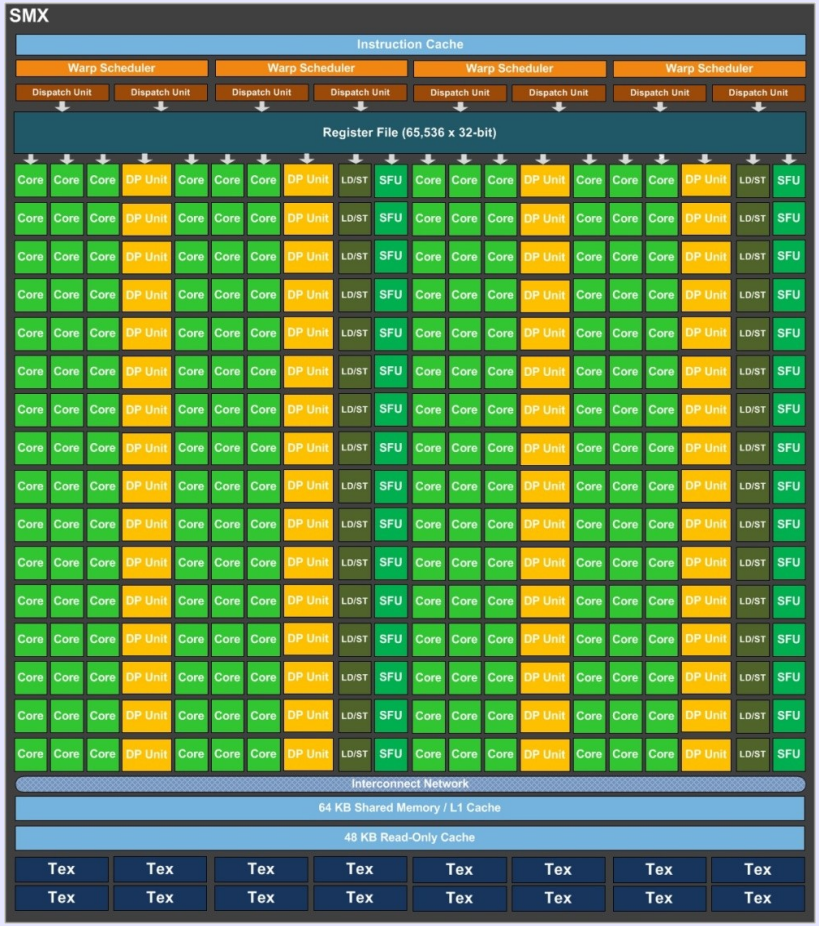
15 SMX, в каждом:

- 192 cuda cores
- 64 double precision
- 32 special function
- 32 банка памяти
- 4 планировшика



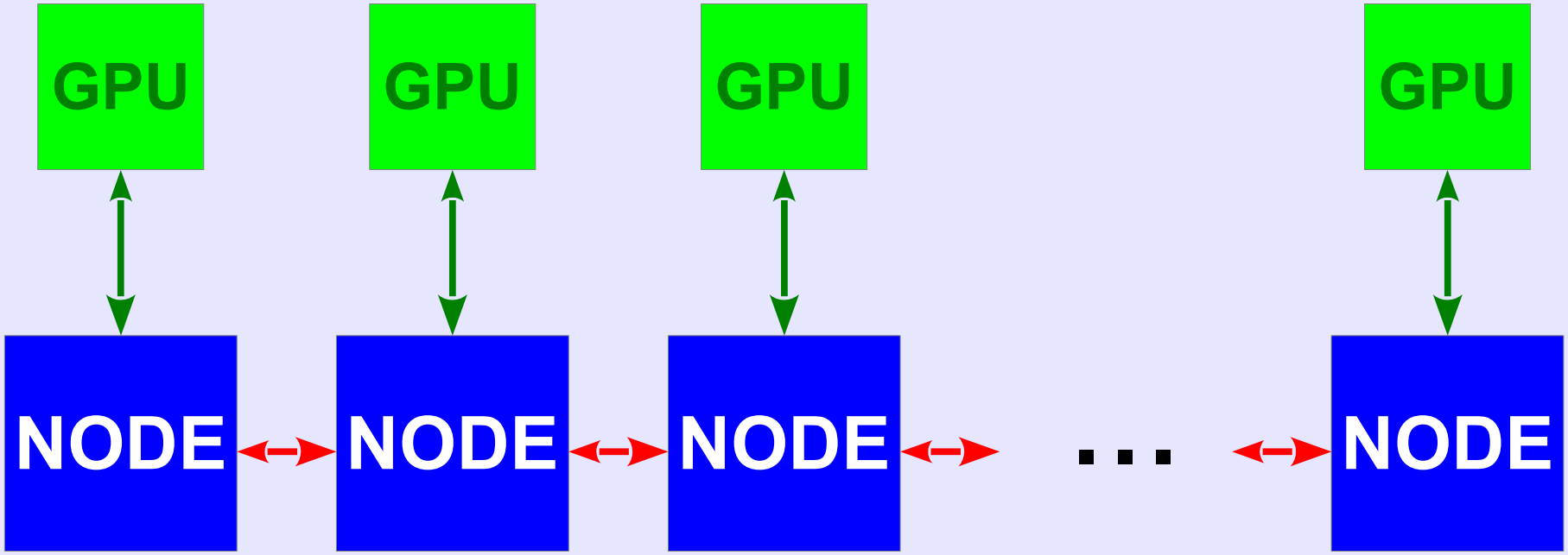
# Типы памяти и расположение

- глобальная (DRAM)
- текстурная (DRAM)
- локальная (DRAM)
- константная (DRAM)
- разделяемая (SM)
- регистры (SM)



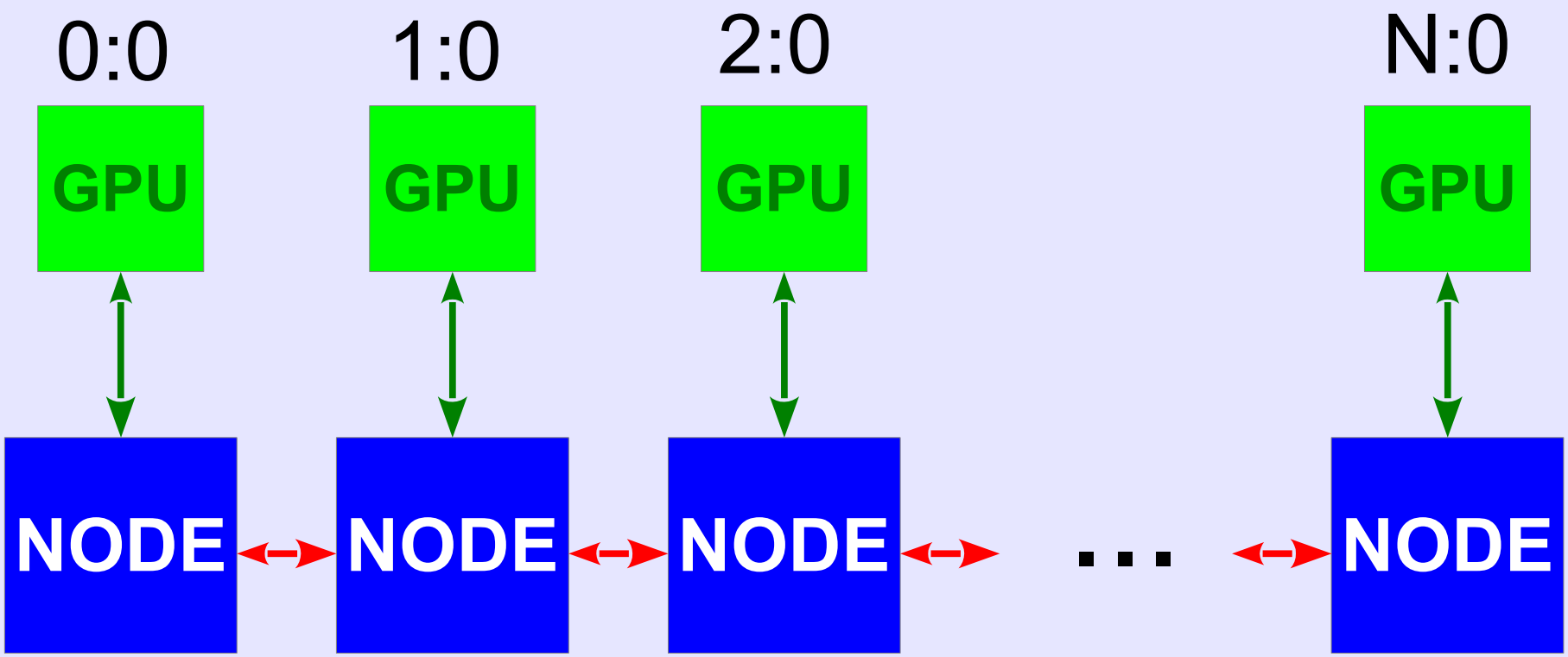
# MultiGPU системы: 1 GPU на ноду

```
cudaMemcpy(d_dest, h_src, size_bytes, cudaMemcpyHostToDevice)
```



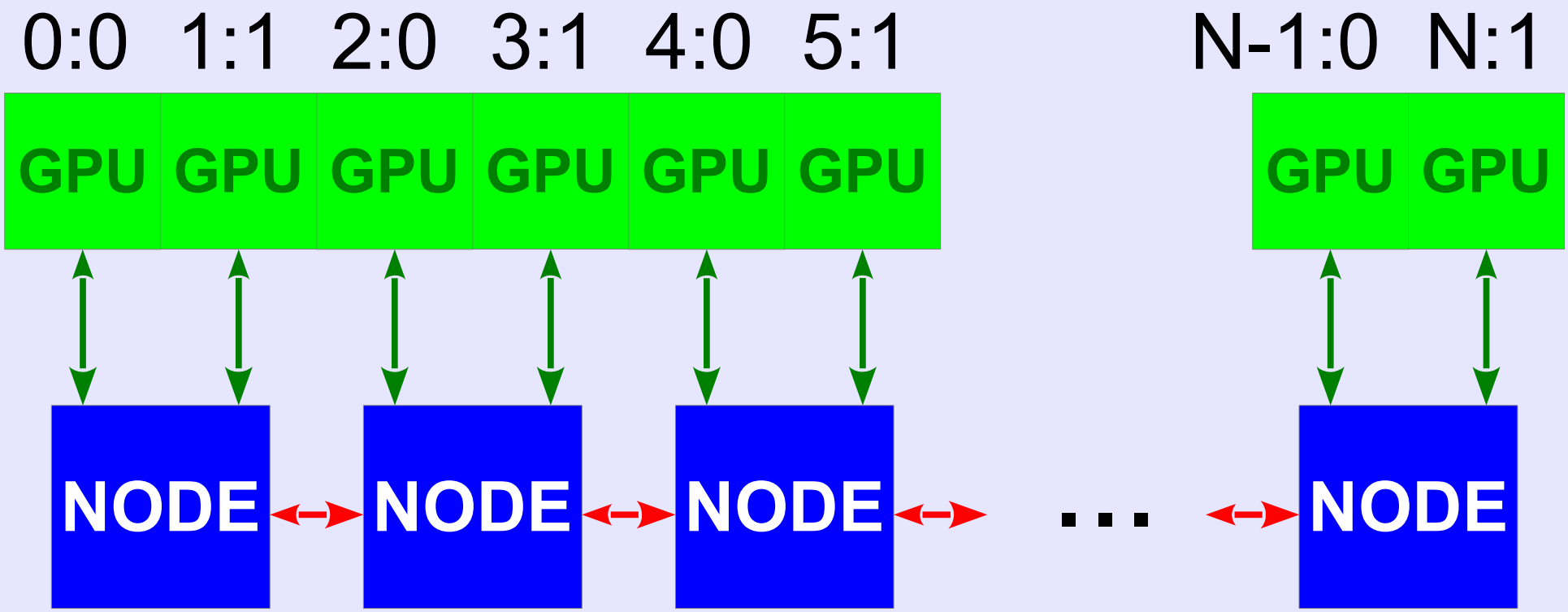

```
MPI_Isend(array_src, array_size, MPI_FLOAT, rank_where,
                                MPI_TAG, MPI_COMM_WORLD, &request);
MPI_Recv (array_dst, array_size, MPI_FLOAT, rank_from,
                                MPI_TAG, MPI_COMM_WORLD, &status);
```

# MultiGPU системы: 1 GPU на ноду



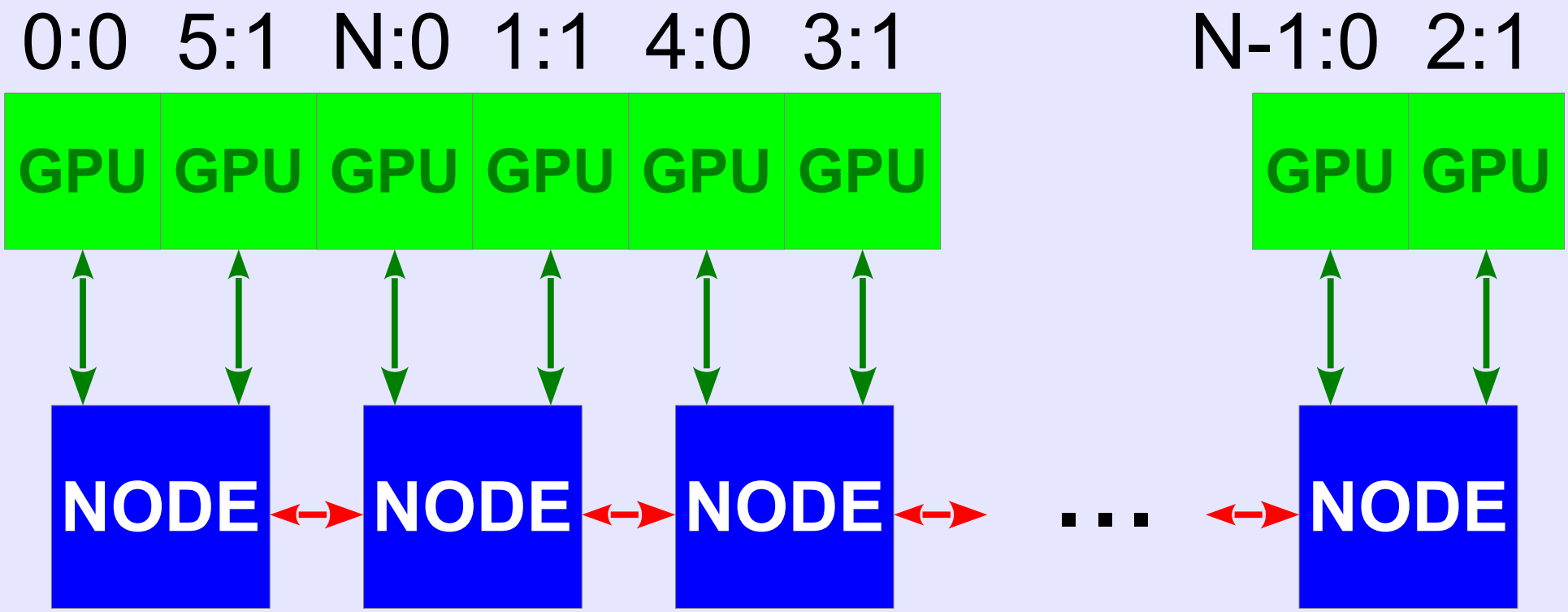

RANK: GPU #

# MultiGPU системы: >1 GPU на ноду




GPU # = RANK%2

# MultiGPU системы: >1 GPU на ноду




GPU # ?= RANK

# Задание CUDA устройства

```
void setDevice(int gpuDevice) {
    int cudaDeviceCount;
    cudaGetDeviceCount(&cudaDeviceCount);

    if (!cudaDeviceCount) {
        std::cout << "No CUDA-capable
devices" << std::endl;
        exit(101);
    } else {
        gpuDevice %= cudaDeviceCount;
        cudaSetDevice(gpuDevice);
    }
}
```

# Распределение GPU устройств

```
void  assignDeviceToProcess() {
  int rank = MPI::COMM_WORLD.Get_rank();
  int size = MPI::COMM_WORLD.Get_size();

  char **hostnames;
  hostnames = new char*[size];
  for (int i=0; i<size; ++i)
    hostnames[i] = new
              char[MPI_MAX_PROCESSOR_NAME];
```

# Распределение GPU устройств

```
int length;
MPI_Get_processor_name(hostnames[rank],
                                  &length);
for (int n=0; n<size; n++)
   MPI_Bcast(hostnames[n],
   MPI_MAX_PROCESSOR_NAME, MPI_CHAR, n,
                                MPI_COMM_WORLD);

int gpuDevice = 0;
for (int n=0; n<rank; n++)
   if (!strcmp(hostnames[n],
                      hostnames[rank]))
      gpuDevice++;
```

# Распределение GPU устройств

```
  setDevice(gpuDevice);

  std::cout << hostnames[rank]
   << " set GPU#" << gpuDevice << std::endl;

  for (int i=0; i<size; ++i)
    delete [] hostnames[i];

  delete [] hostnames;
}
```

# Суперкомпьютер “Ломоносов”

# Характеристики “Ломоносова”

- 480 узлов в gpu partition
- 2x Tesla X2070 на узле
- **6 Gb** DRAM на каждой X2070
- Лимит 3 дня на длительность расчетов
- Каждая C2070 стоит 2k $

# Подключение к “Ломоносову”

1) подключение по ssh для постановки задач:

```
ssh -i Documents/ssh-key/userkey
                user@lomonosov.parallel.ru
```

2) для компиляции задач:

```
ssh -i Documents/ssh-key/userkey
    user@compiler.lomonosov.parallel.ru
```

3) или: 1) +

```
ssh compiler
```

# CUDA+MPI, основные идеи

- CUDA и MPI работают независимо
- рабочий цикл может выглядеть как:
  - загрузка данных с CPU на GPU
  - расчет на GPU
  - выгрузка данных с GPU на CPU
  - обмен между CPU по MPI
  - следующая итерация

# Пример CUDA + MPI

- main.cpp
- Solver.h
- Solver.cpp
- SolverDevice.cu
- Makefile

# Makefile

```
APPNAME = solver
OBJ = Solver.o SolverDevice.o main.o
INCLUDE = -I/usr/local/cuda/include
LIBPATH = -L/usr/local/cuda/lib64 -L/usr/lib64
LIBS = -lcudart
FLAGS = -g -std=c++0x
NVCCFLAGS = --compiler-options -fno-strict-aliasing -use_fast_math
-arch=compute_20 -code=sm_20

all: $(OBJ)
    mpicxx $(FLAGS) $(INCLUDE) $(OBJ) -o $(APPNAME) $(LIBPATH) $(LIBS)

%.o: %.cpp
    mpicxx $(FLAGS) $(INCLUDE) -c $< -o $@

%.o: %.cu
    nvcc $(NVCCFLAGS) -g -c $< -o $@

clean:
    rm -rf src/*.o
    rm -rf src/*/*.o
```

# Основная часть main.cpp

```
int main(int argc, char *argv[]) {
    MPI::Init(argc, argv);

    assignDeviceToProcess();
    cudaDeviceReset();

    Solver solver;

    solver.allocateMemory();
    solver.initMemory();
  solver.calculate();
    solver.getResult();
    solver.releaseMemory();

    MPI::Finalize();
    return 0;
}
```

# SolverDevice.cu

```
namespace SolverDevice {
    __global__ void kernel(float *d_data) {
        int idx = blockDim.x*blockIdx.x + threadIdx.x;
        d_data[idx] += 10;
    }

    void kernelWrapper(dim3 blocksPerGrid,
                       dim3 threadsPerBlock, float *d_data) {
       kernel<<<blocksPerGrid,threadsPerBlock>>>(d_data);
    }
}
```

# Solver.h

```
namespace SolverDevice {
    void kernelWrapper(dim3 threadsPerBlock, dim3
blocksPerGrid, float *d_data);
}
class Solver {
    float *data, *d_data;
    int nodeRank,totalNodes;
public:
    void allocateMemory();
    void calculate();
    void initMemory();
    void releaseMemory();
    void getResult();
};
```

# Solver.cpp

```
void Solver::allocateMemory() {
    data = new float[SIZE];
    cudaError_t result = cudaMalloc((void **)&d_data, SIZE*sizeof(float));
}
void Solver::initMemory(){
    nodeRank = MPI::COMM_WORLD.Get_rank();
    totalNodes = MPI::COMM_WORLD.Get_size();
    for (int i=0; i<SIZE; ++i)
        data[i] = nodeRank;
    cudaError_t result = cudaMemcpy(d_data, data, SIZE*sizeof(float),
                                                        cudaMemcpyHostToDevice);
}
void Solver::releaseMemory() {
    cudaFree(d_data);
    delete [] data;
}
void Solver::getResult() {
    printf("%d - %f\n", nodeRank, data[0]);
}
```

# Основная часть Solver.cpp

```
void Solver::calculate() {
    SolverDevice::kernelWrapper(32, SIZE/32, d_data);
    cudaError_t result = cudaMemcpy(data, d_data,
                    SIZE*sizeof(float), cudaMemcpyDeviceToHost);

    float *data_temp = new float[SIZE];
    int nextRank = (nodeRank + 1)%totalNodes;
    int prevRank = (nodeRank - 1 + totalNodes)%totalNodes;

    MPI_Status status;
    MPI_Sendrecv(data, SIZE, MPI_FLOAT, nextRank, 0,
                 data_temp, SIZE, MPI_FLOAT,
                 prevRank, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);

    for (int i=0; i<SIZE; ++i)
        data[i] = data_temp[i];

    delete [] data_temp;
}
```

# Эволюция **data** в примере

- **Init**: for (int i=0; i<SIZE; ++i) data[i] = nodeRank;
- **GPU**: d_data[idx] += 10;
- **MPI**: MPI_Sendrecv(data, ..., **nextRank**, ...);

- 0 1 2 3
- 10 11 12 13
- 13 10 11 12

# Компиляция и постановка в очередь

- `$ make`
- `$ sbatch -p gpu -N 5 -n 10 ompi ./solver`

  `Submitted batch job 267066`

# slurm-267066.out

$ cat slurm-267066.out

node6-187-05 set GPU#0

node6-187-05 set GPU#1

node6-187-03 set GPU#0

node6-187-03 set GPU#1

node6-187-02 set GPU#1

node6-187-02 set GPU#0

node6-187-01 set GPU#0

node6-187-01 set GPU#1

node6-187-04 set GPU#0

node6-187-04 set GPU#1

```
printf("%d - %f\n", nodeRank, data[0]);
```

9 - 18.000000

1 - 10.000000

0 - 19.000000

5 - 14.000000

4 - 13.000000

3 - 12.000000

2 - 11.000000

6 - 15.000000

7 - 16.000000

8 - 17.000000

# CUDA 5.0: GPU Direct

- утверждают, что можно будет обойтись без MPI для работы по сети
- прямое копирование между несколькими GPU на одной PCIe
- ускорение работы в результате прямого доступа к памяти GPU с CPU

*“Пока не увижу – не поверю.*

*Звучит слишком круто.”*

# Заключение

- CUDA ускоряет вычисления, но не всегда хватает одной GPU
- для MultiGPU удобно использовать MPI
- CUDA и MPI ортогональны
- поэтому так просто реализовывать и поддерживать
- при этом получается универсальный код
- с минимальными потерями производительности

Вопросы?

# Спасибо за внимание!

Дмитрий Бикулов
dmitry@bikulov.org
Физический факультет МГУ
кафедра компьютерных методов физики