Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики».

Лабораторная работа №3
по теме «Реализация исполняемого процессора и проектирование архитектуры и микроархитектуры нейросетевого процессора»
по дисциплине «Организация вычислительных систем»

Выполнили:
студенты группы P4119
Алексеев Д.И.
Орлова А.В.

Проверил:
Быковский С.В.

Санкт-Петербург
2025

Задание

1. Разработать и реализовать архитектуру вычислителя, предназначенного для моделирования работы нейронной сети, полученной по итогам выполнения лабораторной работы №1 и №2. Рассмотреть варианты распараллеливания и конвейеризации вычислений.

2. Реализовать и разработать формат описания сети и алгоритм конфигурирования/программирования процессора.

3. Сформировать отчет по работе, в котором описать:

3.1. Структуру выбранной сети.

3.2. Архитектуру и реализацию разрабатываемого нейросетевого процессора, включая варианты распараллеливания и конвейеризации вычислений

3.3. Оценку времени вычисления выхода сети на ресурсах процессора.

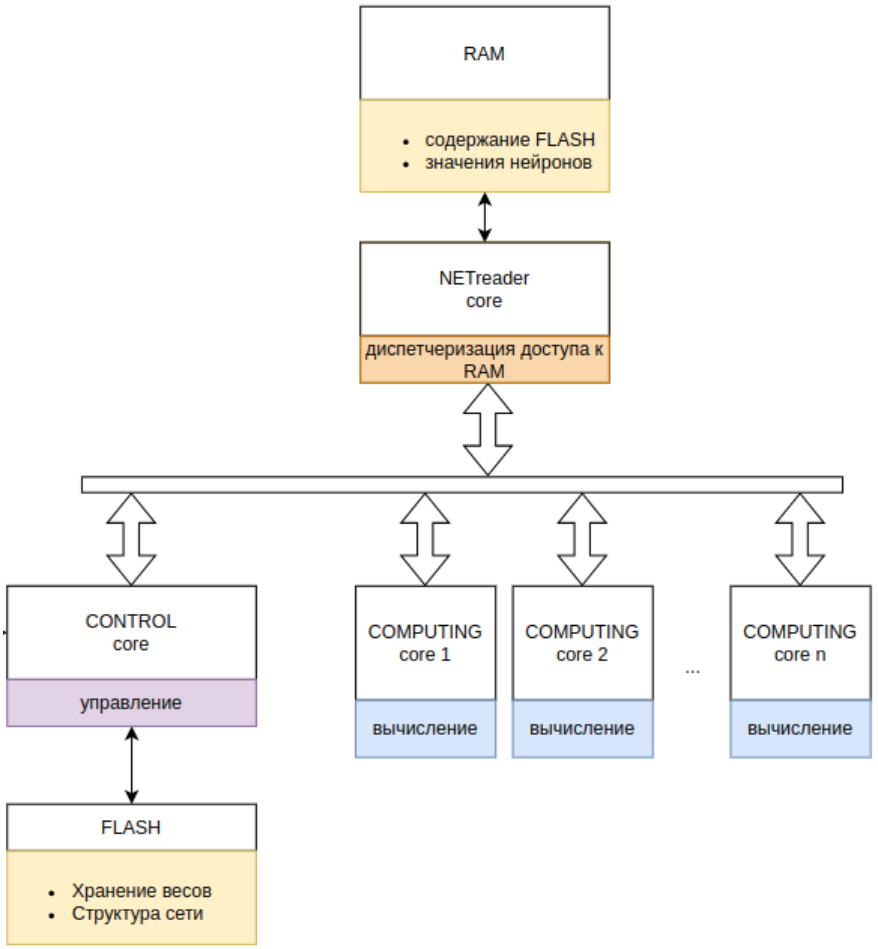
3.4. Оценку загрузки каждого вычислительного ядра при вычислении выхода сети.

Структура выбранной сети

В ходе выполнения первой лабораторной перспективной была полносвязная сеть 49-14-3.

Архитектура нейросетевого процессора

Разрабатываемая система имеет следующую архитектуру:



Краткое описание модулей

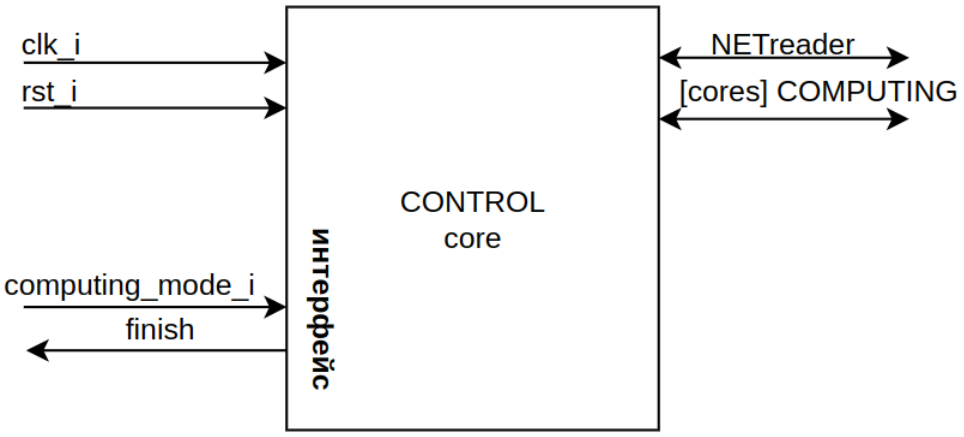
- Вычисления производятся на множестве ядер COMPUTING, которыми управляет CONTROL.
- Модуль RAM используется как память с более быстрым доступом.
- FLASH память позволяет хранить структуру и конфигурацию сети (веса связей), добавляя возможность изменять эти параметры перезаписью.
- NETreader необходим в качестве арбитра общей памяти, к которой обращаются CONTROL и COMPUTING.

Функциональных узлы

CONTROL core

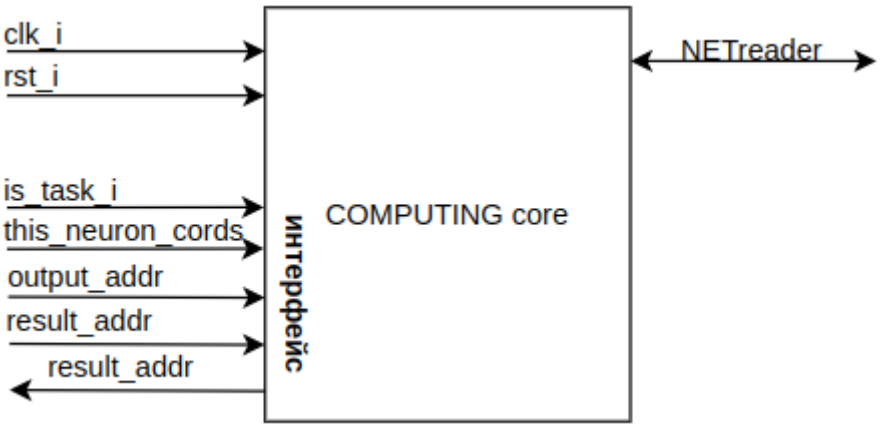
Система предполагает работу в двух режимах/сценариях - вычисление прямого хода сети и реконфигурация сети. Эти процессы могут быть четко представлены в виде конечного автомата. Поэтому микроархитектура узла подразумевает хранение состояния режима работы и текущей стадии выполнения.

Данный узел использует стек для хранения независимых задач, которые могут распределены между вычислительными узлами.



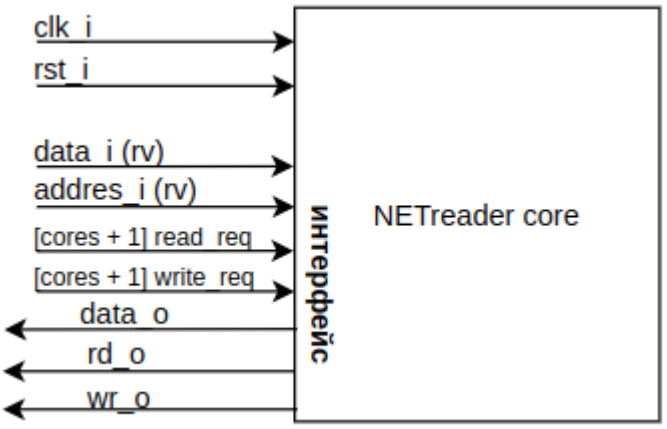
COMPUTING core

Вычислительное ядро скорее имеет схожий принцип работы - строгое движение по состояниям (чтение данных через NETreader в локальную RAM -> вычисление взвешенной суммы -> вычисление функции активации). В локальной памяти хранится необходимые для вычисления данные (значения нейронов предыдущего узла, их количество, значения весов).



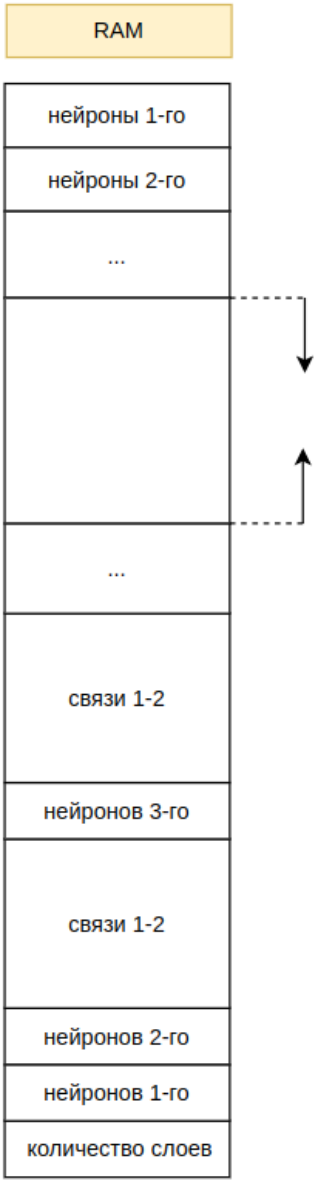
NETreader core

Вычисление адресов значений необходимых для вычисления по идентификатору нейрона. Считывает конфигурацию сети и определяет количества нейронов в предыдущем слое, определяет расположение весов связей. Возможен перенос в состав каждого вычислительного узла.



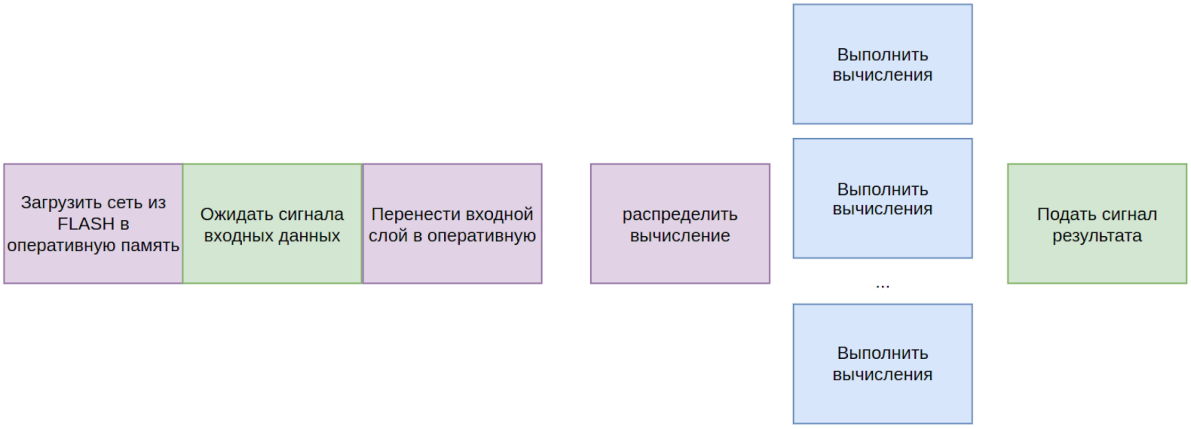
Карта RAM памяти

Хранение структуры сети и значений весов, 32-битная память:



Алгоритм работы процессора

Режим вычисления:



Режим изменения конфигурации:

Детали реализации

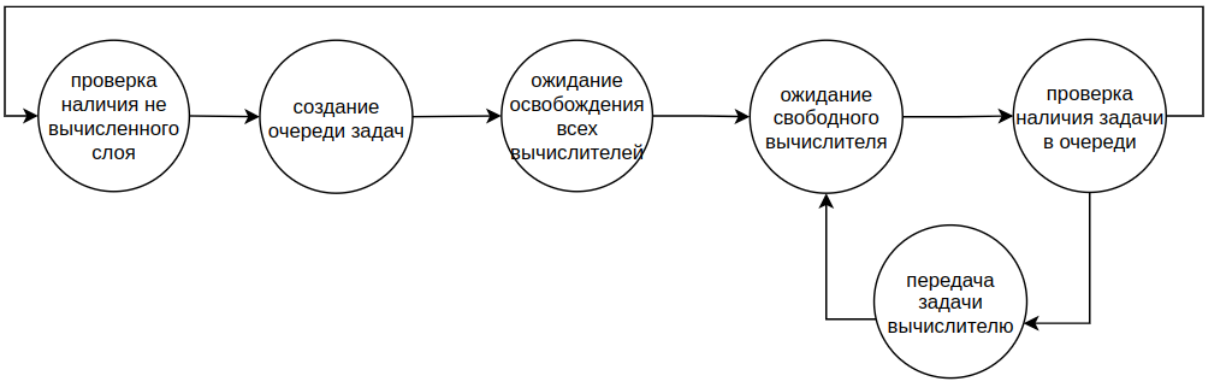
Организация доступа к общей памяти

Выполнена с использования арбитра в виде NETreader, которые выделяет интервалы времени работы каждому подключаемому устройству.

```
void NetReader::poll_requests() {
    is_your_discrete[discrete].write(value_:  true);
    for(uint16_t i = 0; i < 3; ++i) {
        if(read_reqs[discrete].read()) {
            handle_read_req();
            is_your_discrete[discrete].write(value_:  false);
            return;
        } else if(write_reqs[discrete].read()) {
            handle_write_req();
            is_your_discrete[discrete].write(value_:  false);
            return;
        }
        wait();
    }
```

```
    is_your_discrete[discrete].write(value_:  false);
}

void NetReader::execute() {
    while(true) {
        if(rst_i.read()) {
            rd_o.write(value_:  false);
            wr_o.write(value_:  false);
            data_o.write(value_:  0);
            discrete = 0;
            for(uint16_t i = 0; i < CORE_NUMBER + 1; ++i) {
                is_your_discrete[i].write(value_:  false);
            }
        } else {
            if(!(rd_o.read() || wr_o.read())) {
                poll_requests();
                discrete++;
                if(discrete == CORE_NUMBER + 1) discrete = 0;
            }
        }
        wait();
    }
}
```

Конечный автомат CONTROL core

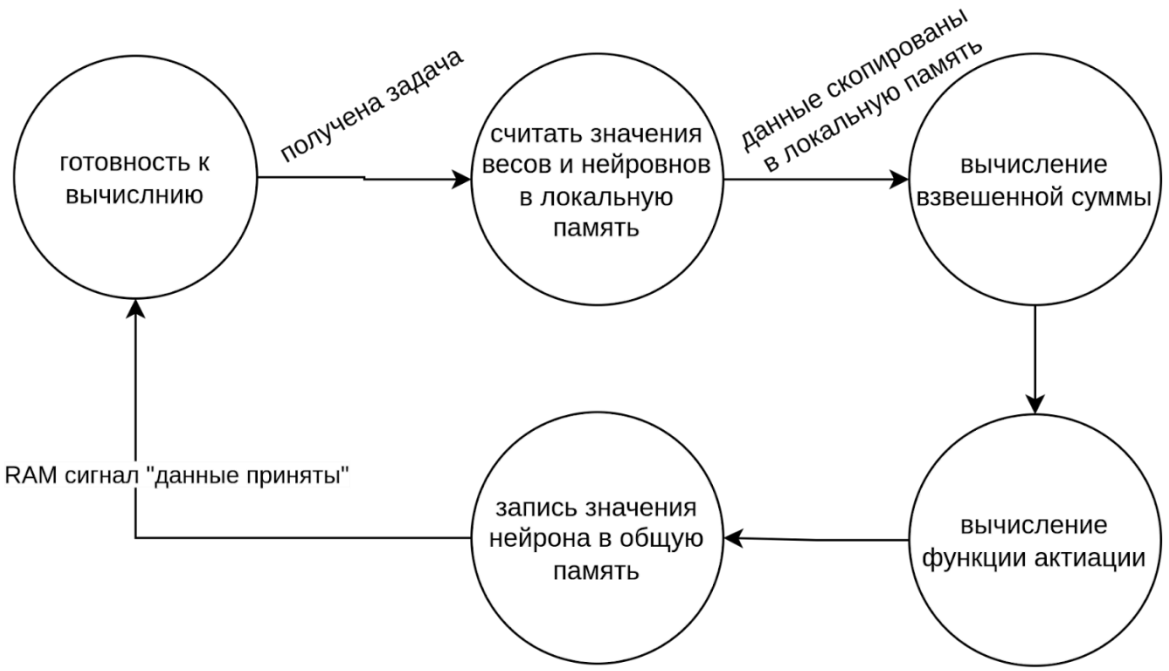


Реализация автомата в C++:

```
void ControlCore::fsm_controller() {
    while (true) {
        if (rst_i.read()) {
            state = IDLE;
            done_layers = 0;
            layers.clear();
        }
        switch (state) {

        case IDLE:
            if (computing_mode_i) {
                std::cout << "starting\n";
```

```
                read_net();
                done_layers = 1;
                state = CHECKING_LAYER;
                finish.write(value_:  false);
            }
            break;
        case CHECKING_LAYER:
            if (done_layers == layers.size()) {
                state = SHOWING_RESULT;
            } else {
                state = CREATING_TASK_Q;
                create_task_q();
            }
            break;
        case CREATING_TASK_Q:
            state = WAITING_FREE_COMPUTING_CORE;
            break;
        case WAITING_FREE_COMPUTING_CORE:
            for(size_t i = 0; i < CORE_NUMBER; ++i) {
                if(core[i]->is_finished.read() == true) {
                    state = CHECKING_ANY_TASK_IN_Q;
                    free_c = i;
                }
            }
            break;
        case CHECKING_ANY_TASK_IN_Q:
            if (q.empty()) {
                bool is_all_free = true;
                for(size_t i = 0; i < CORE_NUMBER; ++i) {
                    if(core[i]->is_finished.read() == false) {
                        is_all_free = false;
                    }
                }
                if(is_all_free) {
                    done_layers++;
                    state = CHECKING_LAYER;
                }
            } else {
                state = ASSIGNING_TASK_TO_COMPUTING_CORE;
                assign_task();
            }
            break;
        case ASSIGNING_TASK_TO_COMPUTING_CORE:
            state = WAITING_FREE_COMPUTING_CORE;
            finish.write(value_:  true);
            break;
        case SHOWING_RESULT:
            show_result();
            std::cout << sc_time_stamp() << "\n";
            wait();
            state = IDLE;
            break;
        }
        wait();
    }
}
```

Конечный автомат COMPUTING core



Реализация автомата в C++:

```
void ComputingCore::fsm_controller() {
    while (true) {
        if (rst_i.read()) {
            ram_addr.write(value_: sc_lv<32>(init_value: 'Z'));
            ram_data_write.write(value_: sc_lv<64>(init_value: 'Z'));
            alu_req.write(value_: false);
            is_finished.write(value_: true);
        }

        switch (state) {
        case IDLE:
            if (is_task_i) {
                this_neuron_cords = this_neuron_cords_i;
                prev_layer_address = prev_layer_address_i;
                result_address = res_address_i;
                rd_completed.write(value_: false);
                state = READING;
                activation = 0;
                alu_reset.write(value_: true);
            }
            break;

        case READING:
            is_finished.write(value_: false);
            read_ram_layer();
            alu_reset.write(value_: false);
            sum_completed.write(value_: false);
            state = COMPUTING_SUM;
            compute_sum();
            break;
        case COMPUTING_SUM:
            if (sum_completed.read()) {
```

```
                state = COMPUTING_ACTIVATION;
                compute_activation();
            }
            break;
        case COMPUTING_ACTIVATION:
            if (activation_completed.read()) {
                state = WRITING;
                write_ram_result();
            }
            break;
        case WRITING:
            if (wr_completed.read()) {
                // printf("TASK[%u][%u] IS COMPLETED! res of neuron: %f\n",
this_neuron_cords >> 16, this_neuron_cords & 0x0000ffff, activation);
                is_finished.write(value_: true);
                state = IDLE;
            }
            break;
        }
        wait();
    }
}
```

Также в COMPUTING core отдельно было рассмотрено АЛУ выполняющее непосредственное вычисление взвешенной суммы. Такое рассмотрение позволяет регулировать количество тактов затрачиваемой на совершение вычисления линейной функции. Также возможно создание еще одного уровня распараллеливания вычислений: т.е. создавать несколько блоков АЛУ в каждом COMPUTING:

```
ALU_linear::ALU_linear(sc_module_name mn)
    : sc_module(nm:  mn), clk_i(name_:  "clk_i"),
      rst_i(name_:  "is_task_i"),
      data_i(name_:  "data_i"),
      weight_i(name_:  "w_i"),
      neuron_i(name_:  "n_i"),
      res_o(name_:  "reso_o"),
      ready_o(name_:  "ready_o")
{
    SC_THREAD(execute);
    sensitive << clk_i.pos();
}

void ALU_linear::execute() {
    while(true) {
        if(rst_i.read()) {
            res_o.write(value_:  0xffff);
            ready_o.write(value_:  true);
            sum = 0;
        } else {
            if(data_i.read()) {
                weight = weight_i.read();
                neuron = neuron_i.read();
                ready_o.write(value_:  false);;
                calculate();
            }
        }
        wait();
    }
}

void ALU_linear::calculate() {
    sum = sum + weight * neuron;
    wait();
    wait();
    res_o = sum;
    ready_o.write(value_:  true);
}
```

Демонстрация

Запуск на одной сети и разном количестве ядер

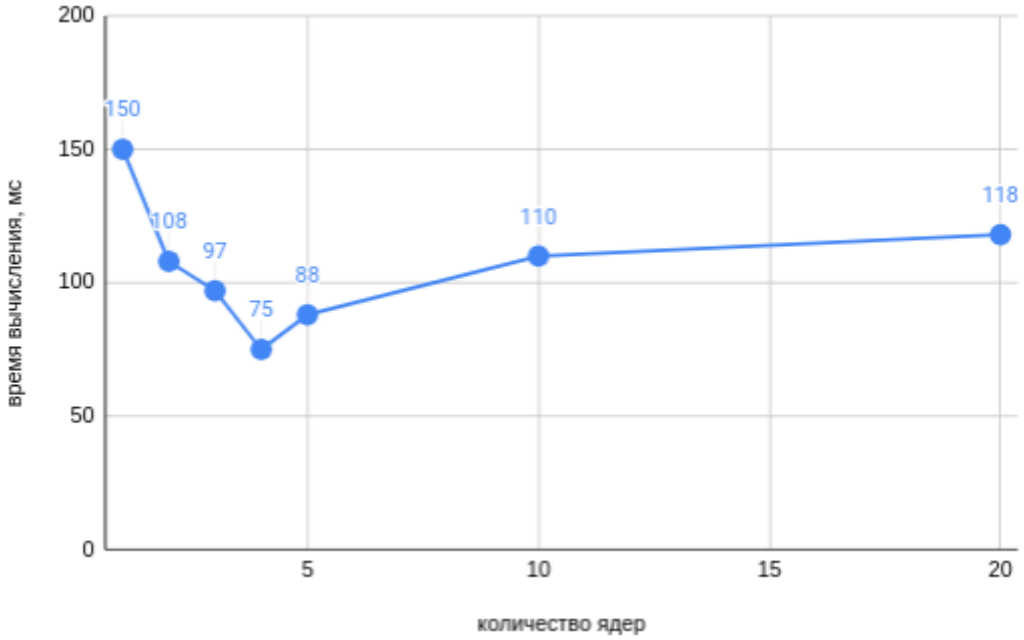
В таблице представлено время работы процессора для вычисления сети 49-14-3 при разном количестве вычислительных ядер.

Количество ядер	Время вычисления при частоте 1 ГГц (мс)
1	150
2	108
3	97

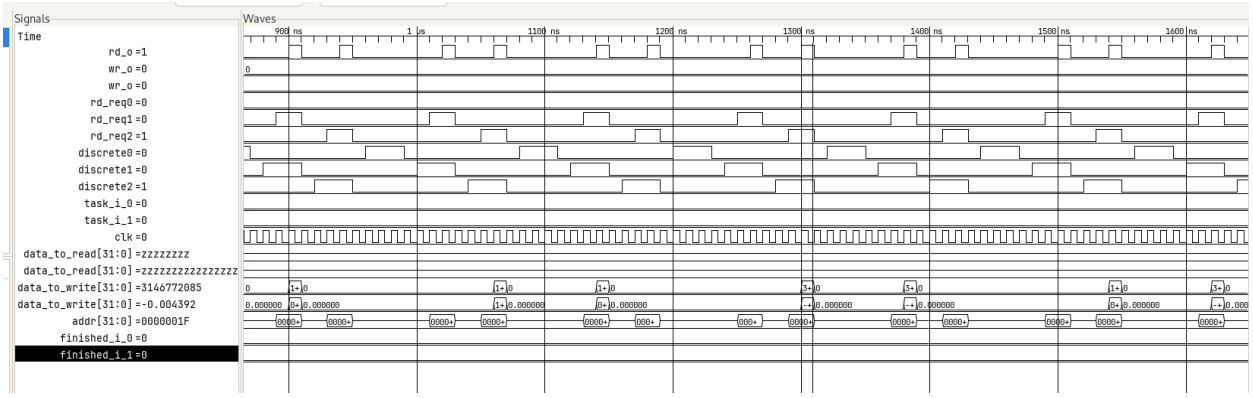
4	75
5	88
10	110
20	118

На рисунке 1 можно наблюдать изменение увеличения числа вычислительных ядер процессора и влияние на количество тактов, необходимых для выполнения вычислений.

Если количество ядер становится больше, то вычисления идут равномерно на каждое ядро, но небольшое использование ядер, является оптимальным решением для данного процессора.



Заметно наличие некоторого оптимального количества ядер: малое количество приводит к повышению времени из-за недостатка вычислительных ресурсов. При сильном увеличении распараллеливания снова происходит времени вычисления из-за накладных расходов множественного обращения к общей памяти.

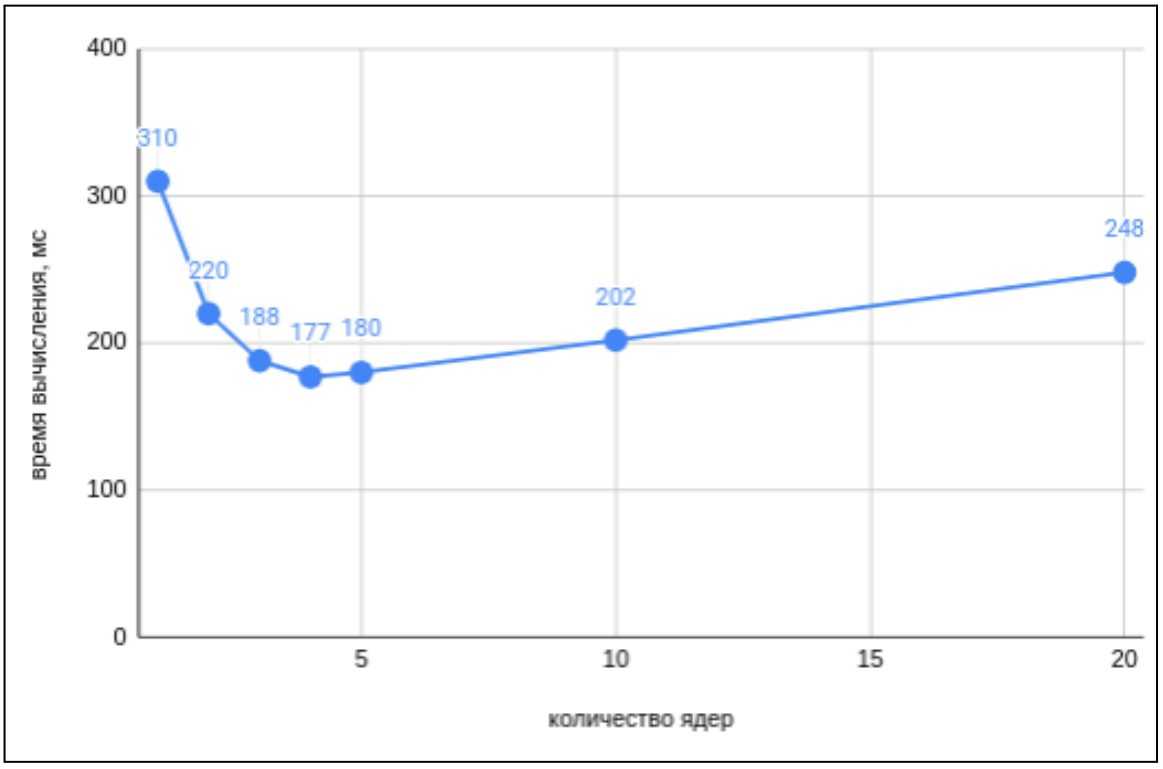


Трассировка сигналов обращения к памяти

Запуск на измененной конфигурации сети

Для использования другой конфигурации сети нужно только записать другие данные в память: изменять код программы не требуется. Можно не только увеличить количество нейронов в слое, но и добавить больше слоев. Например, вычисление результата сети 49-24-12-3 уже потребует такое время:

Количество ядер	Время вычисления при частоте 1 ГГц (мс)
1	310
2	220
3	188
4	177
5	180
10	202
20	248



Наиболее загруженные узлы системы, это вычислительные узлы (COMPUTING) и узел NETreader.

Вычислительные узлы (COMPUTING) выполняют сложные математические операции, такие как свертка, матричное умножение и функции активации. Нагрузка на эти узлы зависит от сложности нейронной сети, объема входных данных и эффективности распараллеливания операций.

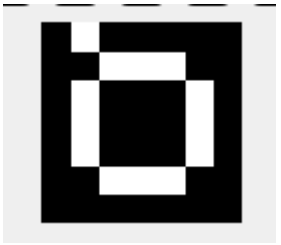
Метрики для оценки загрузки включают среднюю загрузку ядер, пропускную способность шины данных и время выполнения задач.

Узел NETreader отвечает за ввод данных, их предварительную обработку и передачу в вычислительные узлы. При высоких объемах входных данных он может стать узким местом системы. Факторы, влияющие на его загрузку, включают частоту поступления данных, сложность предварительной обработки и эффективность буферизации.

Балансировка нагрузки между узлами системы имеет решающее значение для предотвращения узких мест. Это достигается за счет параллельной обработки данных, динамического распределения задач и использования кэширования.

Результат выполняемой работы:

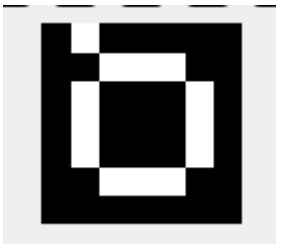
Вывод программы для вычисления сети 49-14-3



```
❱ ./model

        SystemC 2.3.4-Accellera --- Mar  6 2023 09:32:15
        Copyright (c) 1996-2022 by all Contributors,
        ALL RIGHTS RESERVED
number of cores: 4

In file:
/usr/src/debug/systemc/systemc-2.3.4/src/sysc/kernel/sc_object_manager.cpp:153
creating
mem[0] = 3
mem[1] = 49
mem[2] = 14
offset = 3
mem[689] = 3
mem[732] = 65

Info: (I702) default timescale unit used for tracing: 1 ps (wave.vcd)
starting
49 14 3
create_task_q
assign new task [1][0] to 3
assign new task [1][1] to 2
assign new task [1][2] to 1
assign new task [1][3] to 0
assign new task [1][4] to 2
assign new task [1][5] to 0
assign new task [1][6] to 3
assign new task [1][7] to 1
assign new task [1][8] to 2
assign new task [1][9] to 0
assign new task [1][10] to 3
assign new task [1][11] to 1
assign new task [1][12] to 2
assign new task [1][13] to 0
create_task_q
assign new task [2][0] to 3
assign new task [2][1] to 2
assign new task [2][2] to 1
0.679926 0.322922 0.00533704 RESULT: is circle
74830 ns
```

Вывод программы для вычисления сети 49-24-12-3

```
❯ ./model
```



```
        SystemC 2.3.4-Accellera --- Mar  6 2023 09:32:15
        Copyright (c) 1996-2022 by all Contributors,
        ALL RIGHTS RESERVED
number of cores: 4

Warning: (W505) object already exists: controlcore.clk_i. Latter declaration
will be renamed to controlcore.clk_i_0
In file:
/usr/src/debug/systemc/systemc-2.3.4/src/sysc/kernel/sc_object_manager.cpp:153
creating
mem[0] = 4
mem[1] = 49
mem[2] = 24
offset = 3
mem[1179] = 12
mem[1468] = 3
mem[1505] = 0

Info: (I702) default timescale unit used for tracing: 1 ps (wave.vcd)
starting
49 24 12 3
create_task_q
assign new task [1][0] to 3
assign new task [1][1] to 2
assign new task [1][2] to 1
assign new task [1][3] to 0
assign new task [1][4] to 2
assign new task [1][5] to 0
assign new task [1][6] to 3
assign new task [1][7] to 1
assign new task [1][8] to 2
assign new task [1][9] to 0
assign new task [1][10] to 3
assign new task [1][11] to 1
assign new task [1][12] to 2
assign new task [1][13] to 0
assign new task [1][14] to 3
assign new task [1][15] to 1
assign new task [1][16] to 2
assign new task [1][17] to 0
assign new task [1][18] to 3
assign new task [1][19] to 1
assign new task [1][20] to 2
assign new task [1][21] to 0
assign new task [1][22] to 3
assign new task [1][23] to 1
create_task_q
assign new task [2][0] to 3
assign new task [2][1] to 2
assign new task [2][2] to 1
```

```
assign new task [2][3] to 0
assign new task [2][4] to 3
assign new task [2][5] to 1
assign new task [2][6] to 2
assign new task [2][7] to 0
assign new task [2][8] to 3
assign new task [2][9] to 1
assign new task [2][10] to 2
assign new task [2][11] to 0
create_task_q
assign new task [3][0] to 3
assign new task [3][1] to 2
assign new task [3][2] to 1
0.835393 0.36605 0.0165543 RESULT: is circle
173630 ns
```

Диаграмма сигналов 4-ядерной конфигурации при вычислении 49-24-12-3