Разработка и реализация алгоритмов динамического планирования нагрузки на гетерогенный мультипроцессор

Выполнил: Беляев Н. А., 1 курс маг. НГТУ
Руководитель: Перепелкин В. А.

Термины

- Будем называть гетерогенным мультипроцессором (ГМП) вычислитель, состоящий из некоторооого числа процессорных элементов (ПЭ), имеющих собственную оперативную память и соединенных высокоскоростной шиной
- Примером гетерогенного мультипроцессора является узел кластера, содержащий CPU и ускоритель Nvidia CUDA

Проблема

- Задача динамического распределения вычислительной нагрузки является сложной задачей системного параллельного программирования
- Разработчик алгоритмов должен быть абстрагирован от задачи динамического распределения нагрузки

Цель работы

- Разработать, реализовать и провести тестирование алгоритмов динамического распределения вычислительной нагрузки между CPU и ускорителем Nvidia CUDA в системе фрагментированного программирвоания LuNA

Обзор существующих систем ПП с поддержкой гетерогенного мультипроцессора

- Будем под распределением ресурсов понимать назначение задач из некого потенциально бесконечного множества на выполнение на вычислителе из конечного множества вычислителей, входящих в состав мультипроцессора

Классификация алгоритмов распределения ресурсов

- Статические — распределение задач на ПЭ известно до выполнения программы
- Динамические — решение о назначении задач для выполнении на некотором ПЭ принимается во время исполнения программы

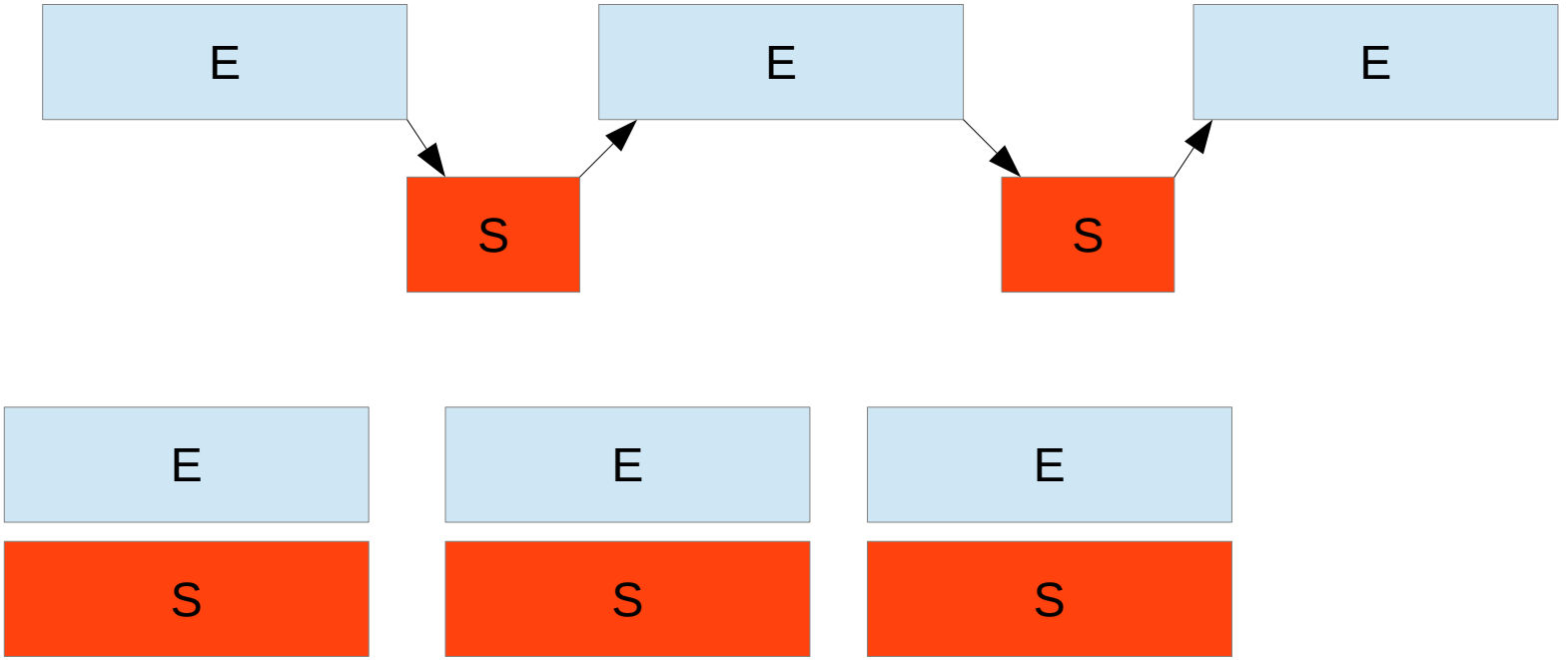
Классификация алгоритмов распределения ресурсов

- Централизованные — задачи распределяются по ПЭ специально выделенными процессорами
- Распределенные — ПЭ забирают задачи из централизованного или распределенного хранилища

Проблемы, связанные с распределением ресурсов

- Сложность принятия решения о распределении ресурсов (как статического, так и динамического)
- Накладные расходы, связанные с выполнением планирования

Планирование на фоне вычислений



Проблемы, связанные с распределением ресурсов

- Проблема экономии траффика: в случае наличия информационных зависимостей между задачами, их целесообразно назначать на выполнение одним вычислителем

Open CL

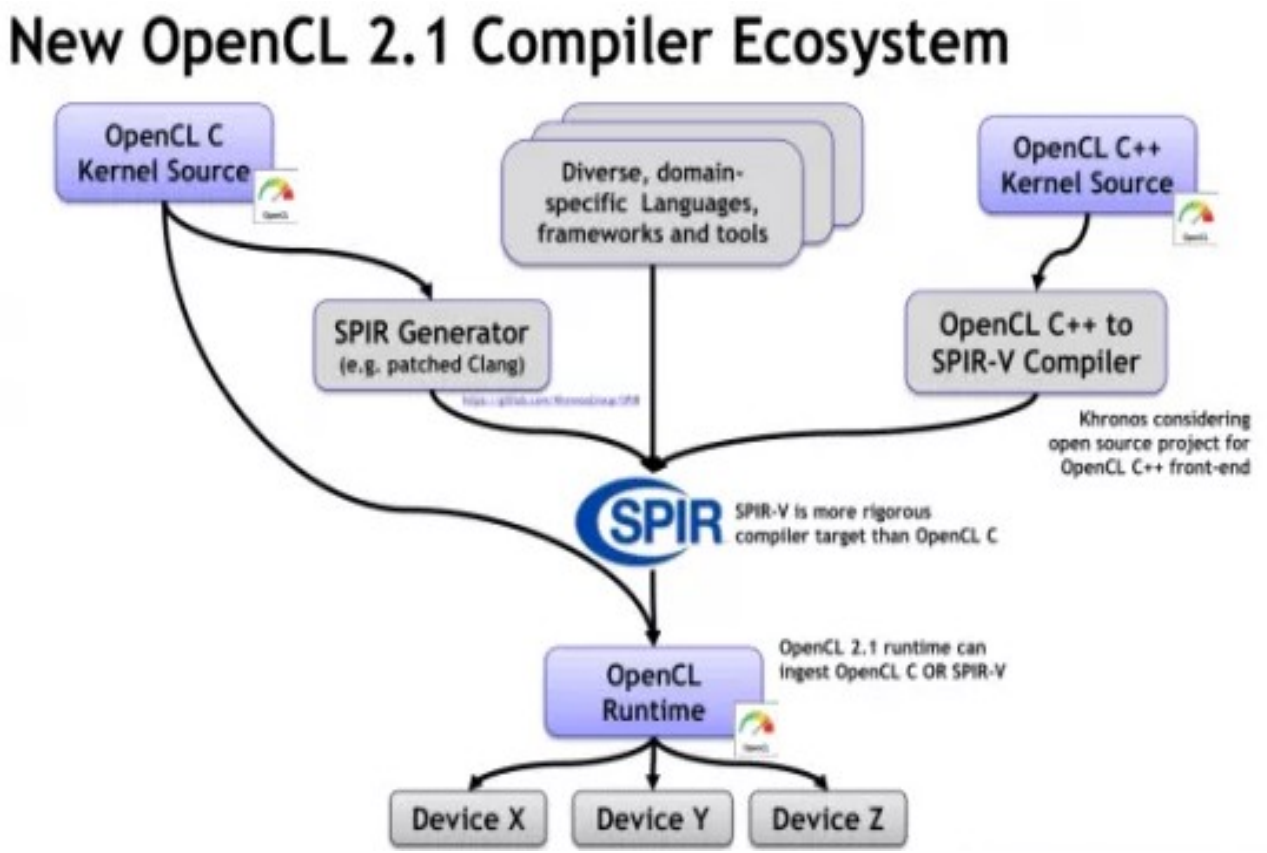
- OpenCL — это открытый стандарт и фреймворк для написания параллельных программ для различных ПЭ (cpu, gpu, fpga, ...)
- В OpenCL входят язык программирования, основанный на стандарте C99 и API

Пример кода OpenCL

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                     NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n, srcA, NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *)
err |= clSetKernelArg(kernel, 1, (void *)
err |= clSetKernelArg(kernel, 2, (void *)

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, k
                             NULL, 0, NU

// read output array
err = clEnqueueReadBuffer(context, memobj
n*sizeof(cl_float),
                          dst, 0, NULL,
```

```
/**
    Testing Kernel. Adds 10 to each element of a given array
*/
__kernel void zeroValues(__global int* values, __global int* ret, int imax)
{
    // thread index and total
    int idx = get_global_id(0);
    int idtotal = get_global_size(0);

    // zero values
    int i;
    for( i = idx; i < imax; i += idtotal)
    {
        ret[i] = values[i] + 10 ;
    }

}
```

OpenCL


New OpenCL 2.1 Compiler Ecosystem
OpenCL C Kernel Source
Diverse, domain-specific Languages, frameworks and tools
OpenCL C++ Kernel Source
SPIR Generator (e.g. patched Clang)
OpenCL C++ to SPIR-V Compiler
Khronos considering open source project for OpenCL C++ front-end
SPIR
SPIR-V is more rigorous compiler target than OpenCL C
OpenCL Runtime
OpenCL 2.1 runtime can ingest OpenCL C OR SPIR-V
Device X
Device Y
Device Z

Решение задачи распределения ресурсов в OpenCL (1)

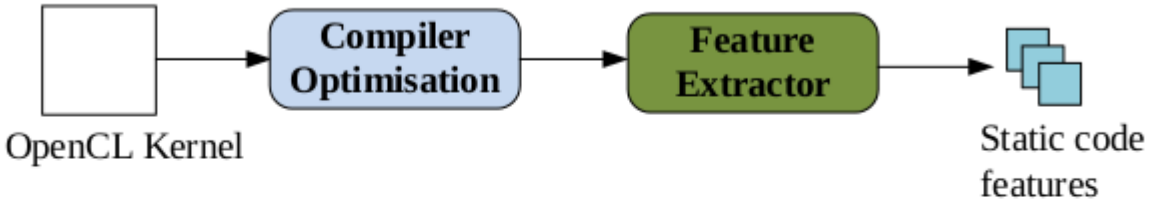


Figure 3. Static code features extraction.

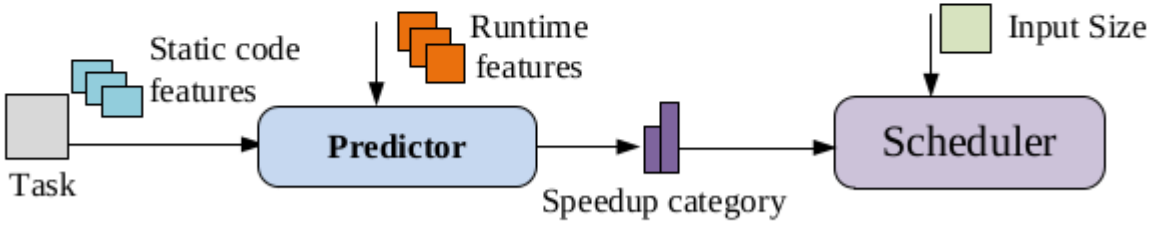
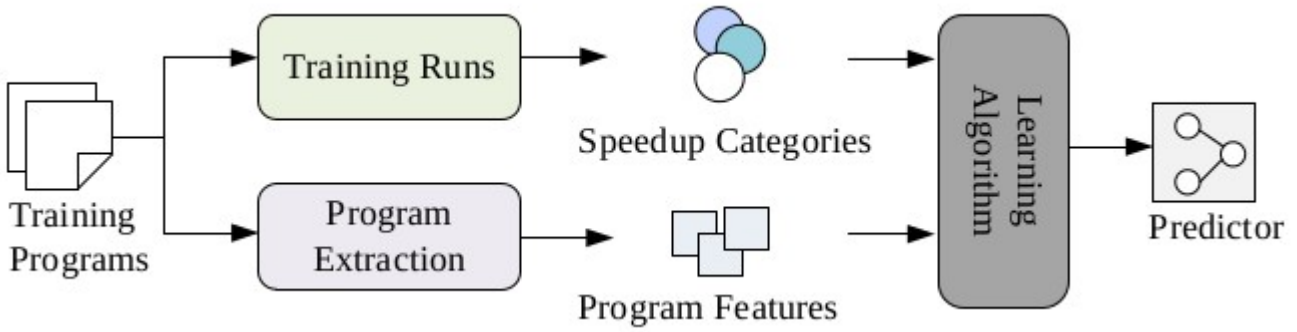


Figure 4. Runtime task speedup prediction



Решение задачи распределения ресурсов в OpenCL (MultiCL)

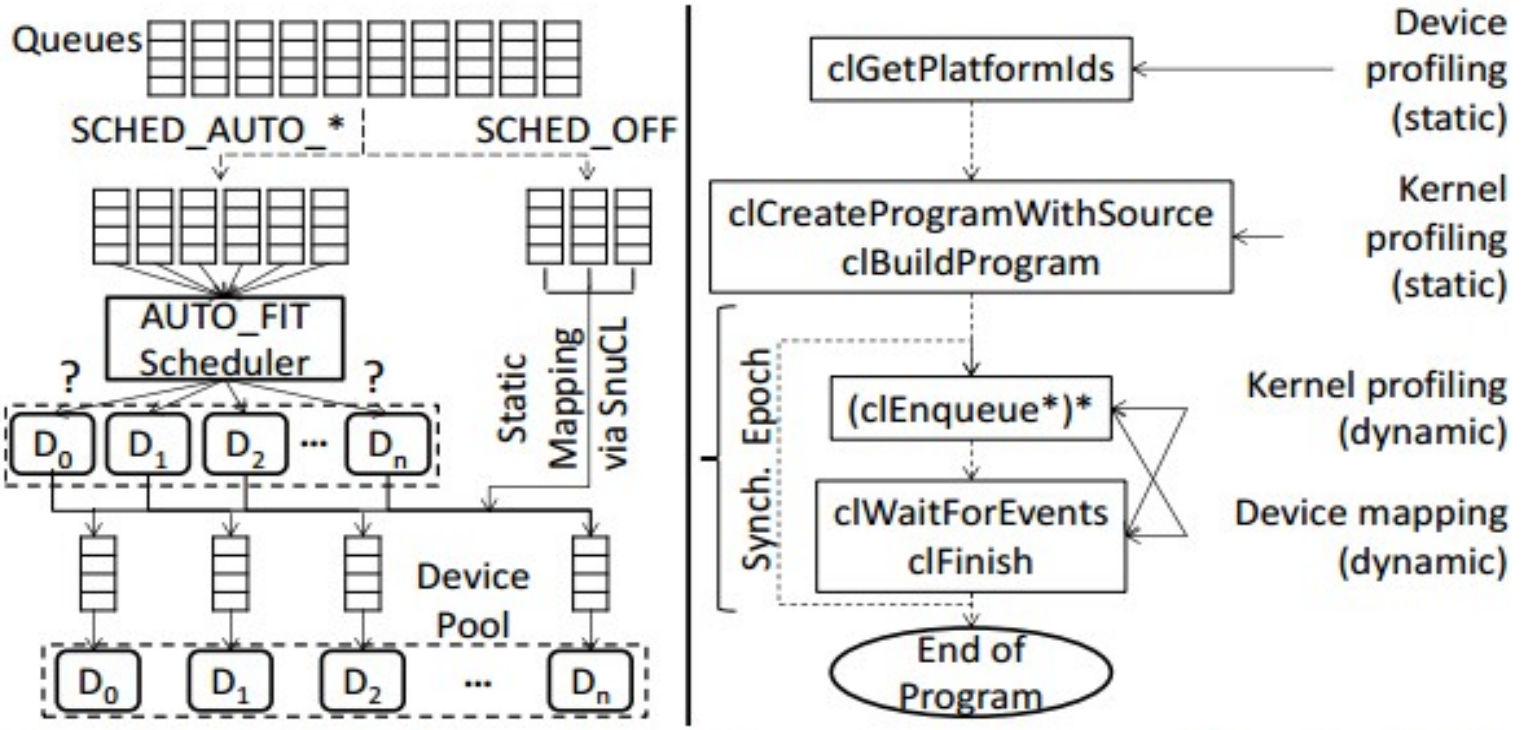
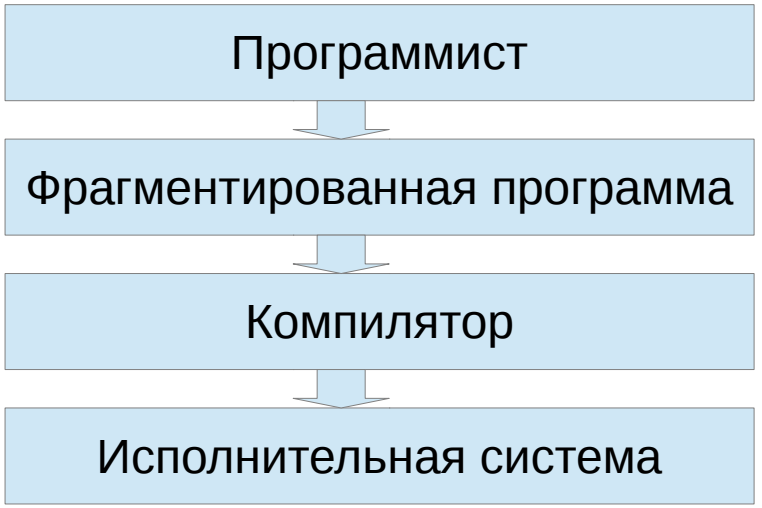


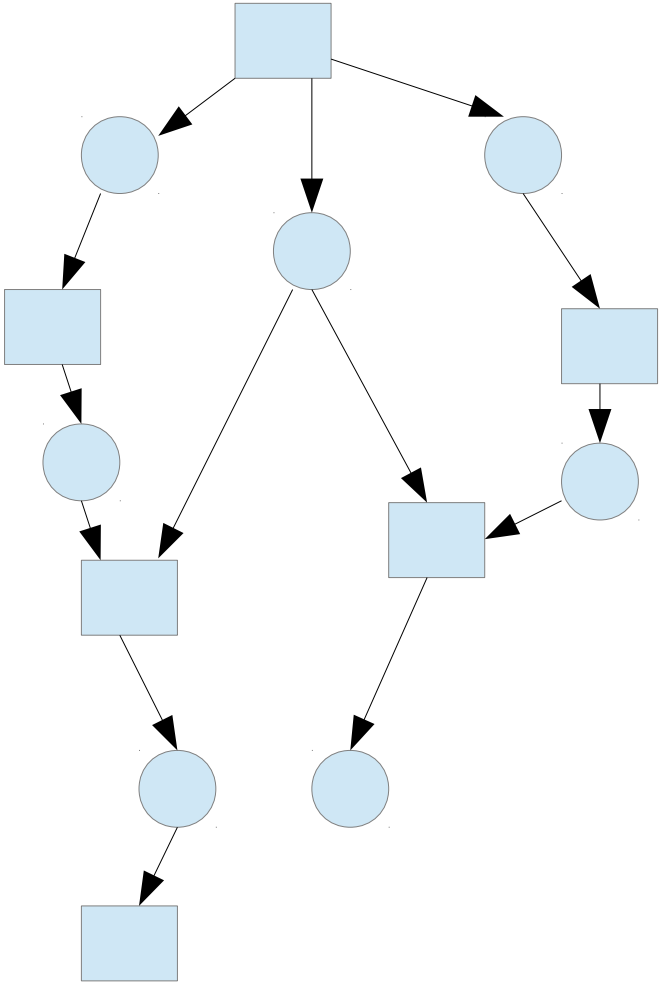
Fig. 1: Left: MultiCL runtime design and extensions to SnuCL. Right: Invoking MultiCL runtime modules in OpenCL programs.

Система LuNA

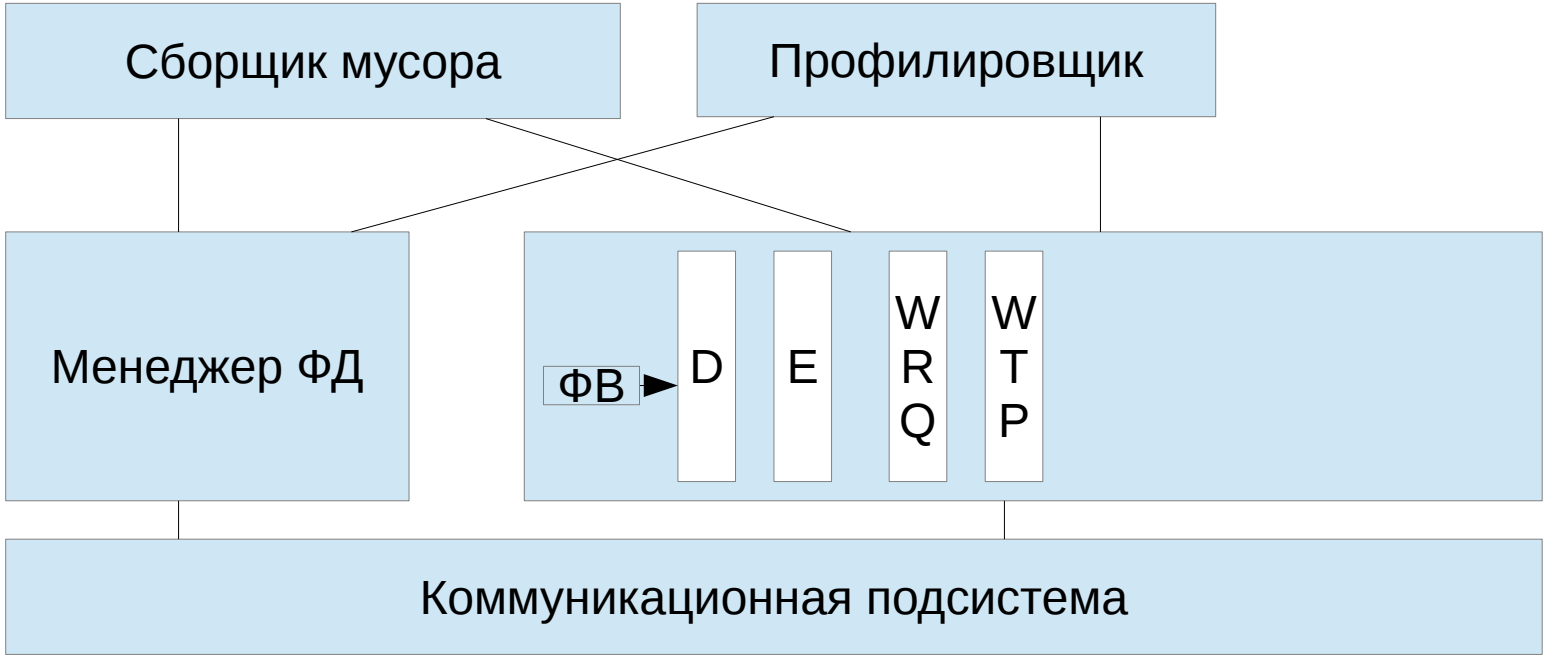
- Система фрагментирвоанного программирования LuNA – это система параллельного программирования, ориентированная на решение задач численного моделирования на высокопроизводительных вычислителях
- Разработчик фрагментированных программ абстрагирован от задач системного программирования и имеет дело непосредственно с алгоритмом, записанным на языке LuNA

Система LuNA

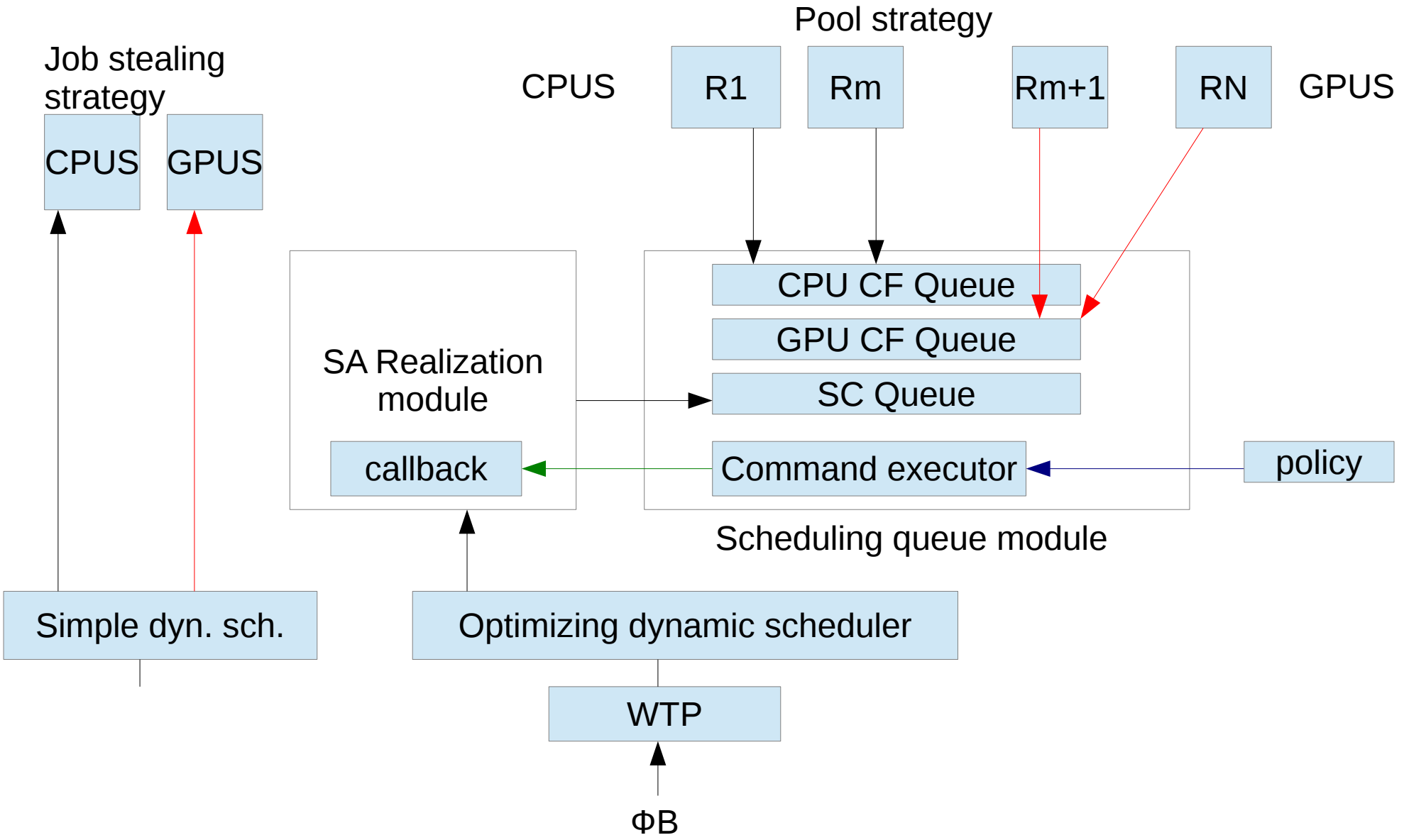




Архитектура системы LuNA



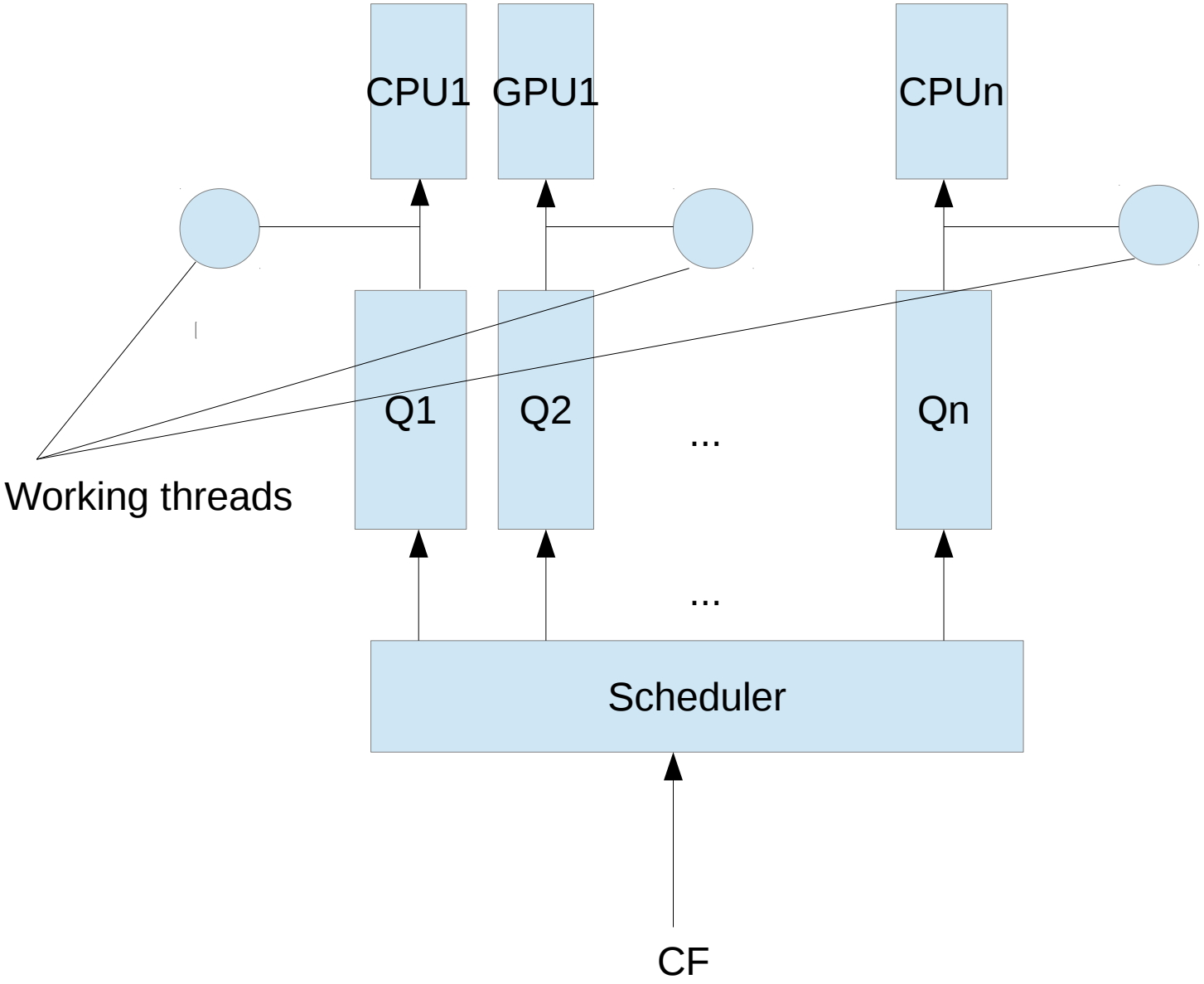
Архитектура исполнительной системы LuNA с поддержкой ГМП



Simple dynamic scheduler

- С каждым ПЭ (ресурсом) ассоциирована очередь задач
- Специально выделенный поток распределяет поступающие в WTP задачи между очередями

Simple dynamic scheduler



Простой алгоритм планирования

- Пусть CFi – текущий ФВ, готовый к исполнению, in(CFi) – множество входных ФД, out(CFi) – множество выходных ФД, Qci- множество ФВ, назначенных на i-й CPU, Qg – множество ФВ, назначенных на GPU, c(CFi) = {1, Cfi реализован для CPU, 0 – иначе

  g(CFi) = {1, Cfi реализован для GPU, 0 – иначе

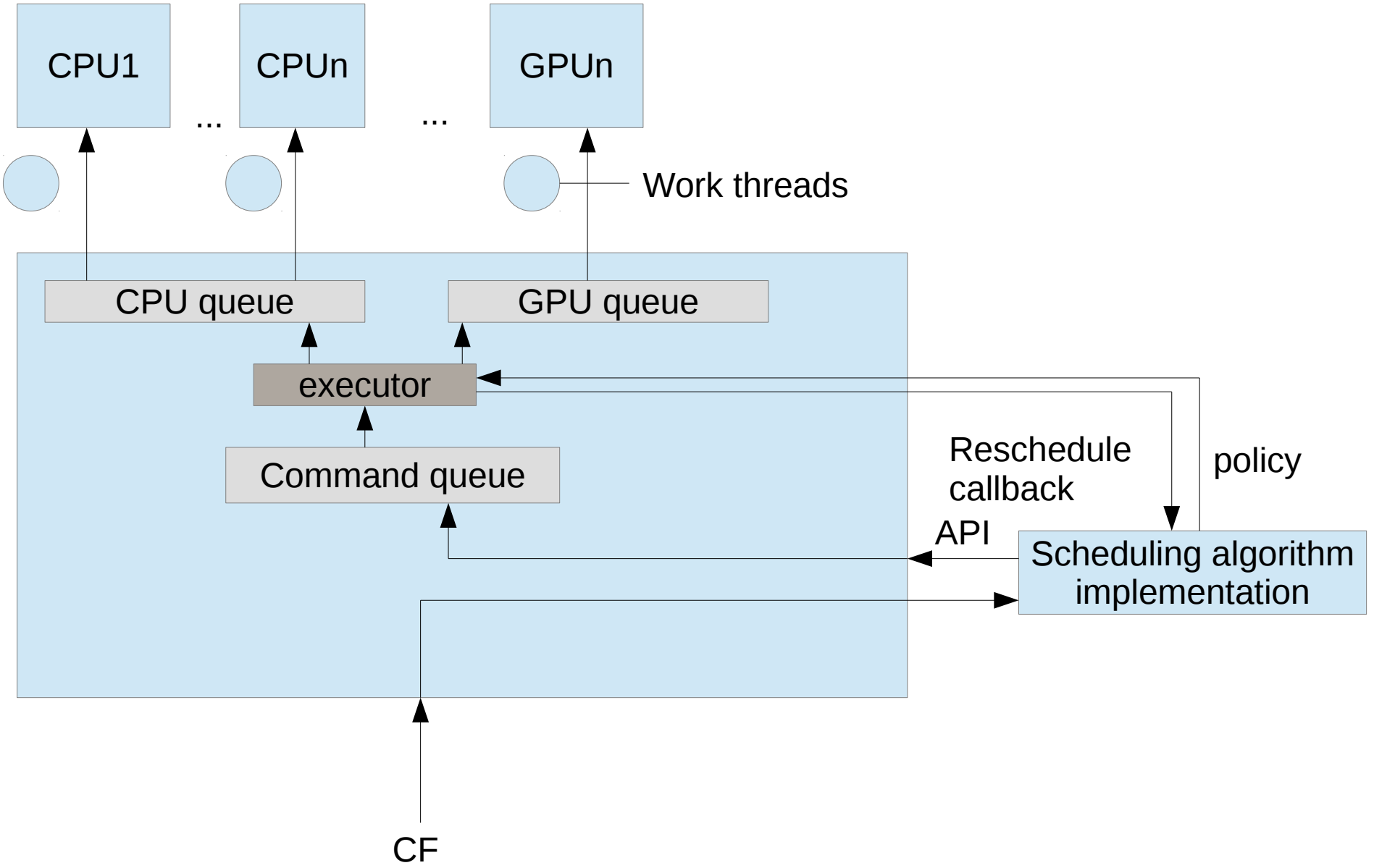
Простой алгоритм планирвоания

```
Procedure schedule (CFi)
begin
    If (!c(CFi) | (Qg =  Ø & g(CFi))) then Qg = insert{Cfi, Qg}
    Else begin
        f = 0
        foreach Qci begin
            if (Qci = Ø) begin
                Qci = insert(Cfi, Qci)
                f = 1
                end
                end
                if (f = 0) then insert(Cfi, Qck), Qck = next_cpu
    endif
end
```

Optimizing dynamic scheduler

- Задачи находятся в общих для всех CPU и GPU очередях
- Реализована возможность непосредственной работы с планом назначения задач на ПЭ

Optimizing dynamic scheduler



Command queue

- Команды:

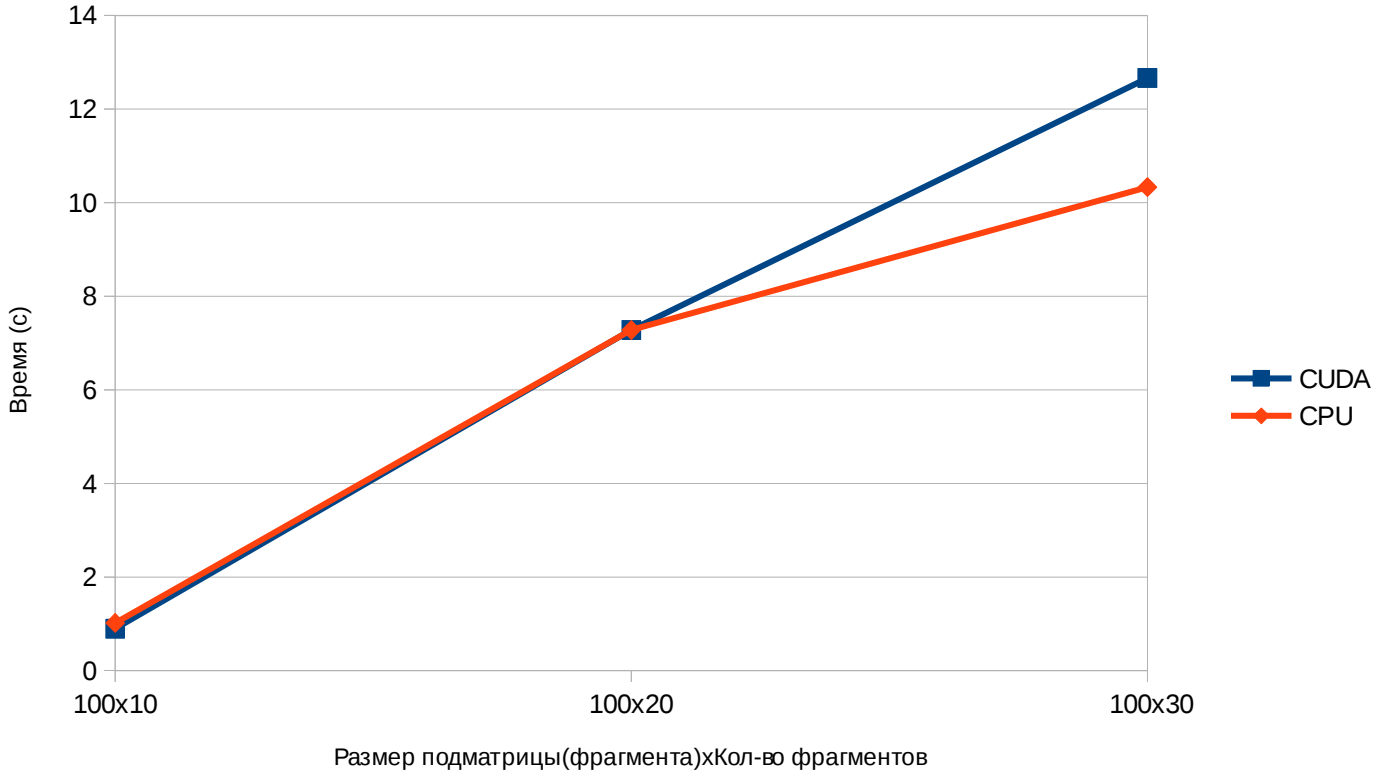
  COPYINS — скопировать все входные ФД для данного ФВ, находящиеся не в точке назначения

  - COPYDF — скопировать ФД
  - SCHEDCFGPU(CPU) — назначить ФВ для исполнения на CPU\GPU
  - FREEDEV — освободить память ПЭ, занимаемую ФД

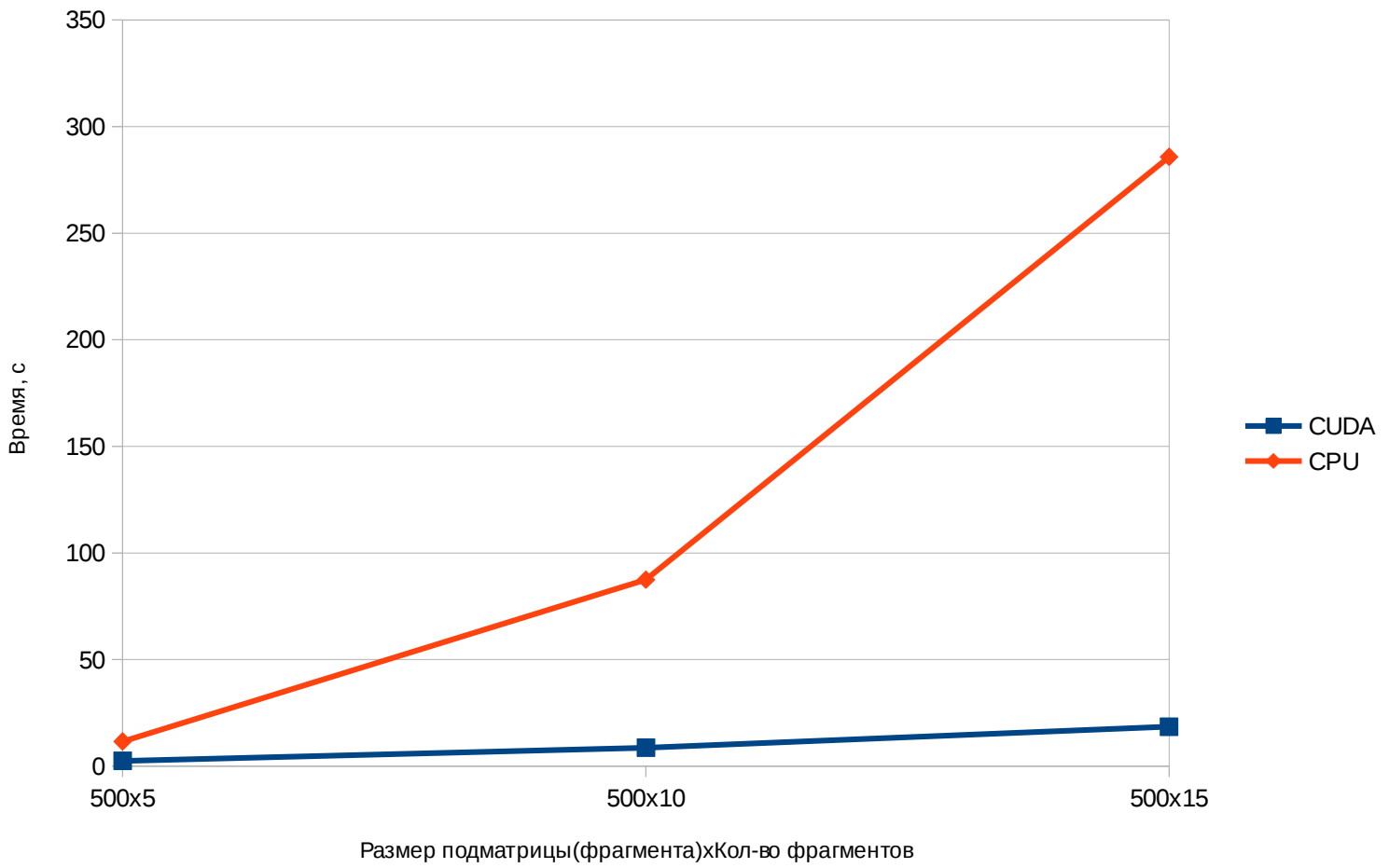
Тестирование

- Тестирование проводилось на машине с CPU 6x Intel Xeon X5660 @2.8GHz + NVIDIA GT200GL [Quadro FX 4800]
- Тестовая задача: умножение матриц
- Алгоритм

Тестирование



Тестирование



Результаты

- Разработана архитектура простого и оптимизирующего планировщиков
- Реализованы соответствующие прототипы
- Проведено тестирование прототипов

Планы

- Реализация интеллектуальных алгоритмов планирования