Подсистема сервисов

Пространство имён: ttgLib::Services

Большая часть библиотеки ttgLib реализована в виде сервисов — интерфейсов, предоставляющих строго заданную функциональность. В качестве примера можно привести такие сервисы как `Logger`, отвечающий за создание записей о работе программы, и `Timer`, выполняющий замеры времени. Каждый из сервисов существует в одном экземпляре, ссылку на который можно получить с помощью статического метода `GetRef()`. Поэтому, например, для получения текущего времени следует вызывать `Timer::GetRef()->GetSeconds()`.

Схема запуска сервисов

Для инициализации сервисов необходимо создать экземпляр класса `ServiceStarter`, в конструкторе которого указать одну из поддерживаемых схем работы. Данные схемы реализованы как следующие статические методы класса `ServiceConfigPresets`:

- `BasicConfig()`. Определяет схему работы без использования внешних утилит и предоставления каких-либо возможностей для подключения к программе извне. Предназначен для случаев, когда не требуется функциональность подсистемы утилит.
- `LocalUtilityConfig()`. Позволяет с помощью утилит подключиться к работающей программе, но только с локального компьютера. Идеален для отладки, или же в целях обеспечения требуемого уровня безопасности.
- `RemoteUtilityConfig()`. Предоставляет возможность подключаться к программе с любого внешнего компьютера через утилиты или web-браузер. В качестве аргумента требуется указать порт, по которому будет «видна» запущенная программа.
- `RemoteWithAuthUtilityConfig()`. В дополнение к предыдущей схеме работы предоставляет функциональность авторизации. Логин и пароль пользователя задаются через аргументы метода.

Таким образом, процесс подключения и инициализации библиотеки ttgLib может имеет следующий вид:

```
#include <ttgLib.h>

using namespace ttgLib::Services;

int main()
{
   ServiceStarter ss(ServiceConfigPresets::BasicConfig());
   //Computing.
}
```

Для случаев, когда требуется дополнительная настройка отдельного сервиса имеется возможность переопределить базовую схему работы. Для этого к ней с помощью оператора «плюс» необходимо добавить настройки отдельного сервиса:

```
ServiceStarter ss(ServiceConfigPresets::BasicConfig() + LoggerConfig("MyLog.txt"))
```

Настройки отдельного сервиса, добавленные как правый операнд, перекрывают предыдущие, тем самым позволяя изменять любые опции базовых схем работы.

Динамические параметры

Пространство имен ttgLib::Parameters

Основным механизмом встраивания возможностей библиотеки ttgLib в оптимизируемые приложения является механизм динамических параметров. Они реализованы в виде шаблонных классов вида `Parameter<T>` и с точки зрения программиста эквивалентны обычным переменным, однако их использование позволяет подсистеме оптимизации собирать статистику и незаметно влиять на программу. А благодаря внешним утилитам пользователь получает возможность не только наблюдать за значениями подобных динамических переменных, но и изменять их даже с удалённого компьютера или web-браузера.

Базовые типы параметров

В настоящий момент в библиотеке реализованы следующие базовые типы параметров:

– `Parameter<T>`. Является основным типом, позволяющим сделать практически любую переменную динамической. Предназначен в первую очередь для предоставления доступа извне для пользовательских полей, реализующих функциональность счётчиков, информационных надписей и т.д.. Примеры:

```
Parameter<int> imax = 10;
Parameter<double> res = 0.0;
//...
for (int i = 0; i < imax; i++)
   res += arr[i];
Parameter<std::string> out = "Finished";
```

– `BoundedParameter<T>`. Данный тип является ограничением основного типа. Другими словами, соответствующая динамическая переменная может принимать значения только из диапазона [lo, hi], где сами значения lo и hi задаются в конструкторе параметра. Предназначен в первую очередь для оптимизационных целей, так как ограничение диапазона значений существенно ускоряет процесс оптимизации. Примеры:

```
BoundedParameter<int> block_size(16, 32);
block_size = 24;
//...
for (int i = 0; i < size / block_size + 1; i++)
   for (int j = 0; j < block_size; j++)
      //...
```

– `EnumParameter<T>`. Версия динамического параметра для перечислений. Позволяет определить перечисление как набор из элементов базового типа, в дальнейшем работа с которым эквивалентна использованию параметра типа `Parameter<T>`, при условии, что присваиваться могут лишь значения из перечисления. Предназначен как для упрощения работы с программой через утилиты (например, задание режимов работы), так и непосредственно при оптимизации. Примеры:

```
EnumParameter<float> en1("{ -1.0, 0.0, 1.0 }");
EnumParameter<int>   en2("{ 0, 1, 2 }");
en1 = 0.0f;
en2 = (int)(en1 + 1.0);
```

Элементы перечисления могут быть заданы двумя способами. Первый вариант заключается в использовании при создании параметра строки вида "`{ value1, value2, value3, valueN }`", где `valueX` является строковой записью соответствующего элемента. Второй вариант — создать STL-контейнер `std::vector<T>`, в который необходимо поместить нужные элементы и который необходимо передать в конструктор параметра. Стоит отметить, что первый вариант применим только для тех случаев, когда базовый тип является стандартным и может быть распознан из строковой записи, в то

время как второй является более универсальным и может быть использован в том числе и для собственных типов. Таким образом, приведённый ранее пример можно переписать следующим образом:

```
std::vector<float> en1Vals;
en1Vals.push_back(-1.0);
en1Vals.push_back(0.0);
en1Vals.push_back(1.0);
EnumParameter<float> en1(en1Vals);
en1 = 0.0f;

std::vector<int> en2Vals;
en2Vals.push_back(0);
en2Vals.push_back(1);
en2Vals.push_back(2);
EnumParameter<int> en2(en2Vals);
en2 = (int)(en1 + 1.0);
```

Также стоит отметить, что можно переопределить строковые имена перечисляемых элементов, под которыми они будут отображаться во внешних утилитах. Для этого необходимо передать в конструктор параметра функцию вида `std::string (*NameConverter)(T)`, которая для заданного элемента возвращает его отображаемое имя. Примеры:

```
std::string NameConverter(int val)
{
   char buf[1024];
   sprintf_s(buf, "Type #%ld", val);
   return buf;
}

//...
std::vector<int> vals;
en2Vals.push_back(0);
en2Vals.push_back(1);
en2Vals.push_back(2);
EnumParameter<int> en(vals, NameConverter);
```

– `ActionParameter`. Данный тип предназначен для обработки событий, которые могут быть инициированы как в процессе оптимизации, так и пользователем из внешних утилит. Полезны для реализации различных сценариев управления программой извне (сохранение промежуточных данных, включение визуализации и т.д..). Примеры:

```
void ProcessEvent(Void)
{ printf("Event has been processed\n"); }
//...
ActionParameter action;
action.OnTriggered() += MakeDelegate(ProcessEvent);
//...
action.Trigger();
```

– `ParameterGroup`. Данный тип предназначен для объединения в группы любых других параметров с целью их систематизации. В утилитах каждая группа параметров будет отображаться как один уровень исходного дерева, а при оптимизации группы применяются для задания параметров, влияющих на производительность только отдельного вычислительного ядра. Примеры:

```
Parameter<int> p1;
BoundedParameter<double> p2(0.0, 1.0);
ActionParameter p3;

ParameterGroup group;
p1.Attach(&group);
```

```
p2.Attach(&group);
p3.Attach(&group);
```

Составные типы параметров

Составные параметры являются обёртками поверх базовых, реализующими наиболее часто используемые конструкции. Данные параметры распознаются внешними утилитами и подсистемой оптимизации, что позволяет задействовать дополнительную априорную информацию для проведения более качественной оптимизации или визуализации. На настоящий момент составные параметры представлены следующими типами:

– `GridParameter1D`. Позволяет задать разбиение произвольного одномерного отрезка на неравномерные части с помощью динамических параметров. В первую очередь предназначен для балансировки загрузки между неоднородными вычислителями и/или узлами. Таким образом, поиск оптимального разбиения может быть легко выполнен подсистемой оптимизации в аналогии со следующим примером:

```
GridParameter1D grid(N);          //constructing grid with N cells
grid.SetBoundaries(0, 1024);
grid.SetAlignment(128);
//...
for (int i = 0; i < grid.GetCount(); i++)
   device[i]->compute(grid[i].GetLowerBoundary(), grid[i].GetUpperBoundary());
```

– `CudaGridParameter1D`. В отличие от предыдущего типа `GridParameter1D`, `CudaGridParameter1D` предназначен для задания такого параметра запуска CUDA-ядер, как количество выполняемых на одном мультпроцессоре нитей, который напрямую влияет на скорость вычислений. Стоит отметить, что функциональность данного параметра можно реализовать, например, через `EnumParameter<UInt32>`, однако допустимые значения сильно зависят от архитектуры GPU, поэтому для упрощения процесса программирования и был введён данный тип.
Для получения текущего количества нитей (или размера блока) необходимо вызвать метод `size_t GetBlockSize()`, при этом возвращаемое значение будет зависеть от потока, в котором был вызван данный метод. Если поток является ассоциированным с виртуальным ttgLib-устройством (что всегда верно при использовании гибридных примитивов), то будет возвращён размер, поддерживаемый данным устройством. В противном случае будут запрошены свойства CUDA-устройства, выбранного в данном потоке с помощью метода `cudaSetDevice()` из CUDA Runtime API. К примеру, при проведении вычислений на ускорителе с CUDA Compute Capability 1.3 максимально возможный размер блока будет равен 512 нитей, в то время как при использовании ускорителя с архитектурой Fermi или Kepler размер блока может равняться 1024 нитям. Примеры:

```
CudaGridParameter1D p;
dim3 threads(p.GetBlockSize());
dim3 grid(N / p.GetBlockSize());

someCudaKernel<<<grid, threads>>>();
```

Для упрощения использования данного параметра был введён вспомогательный метод `std::pair<size_t, size_t> GetGrid(size_t threadCount)`, который позволяет создать такое количество блоков заданного размера, суммарное количество нитей которых гарантированно превысит или будет равным величине `threadCount`. Поэтому предыдущий пример можно переписать следующим образом:

```
CudaGridParameter1D p;
dim3 threads(p.GetGrid(N).second);
dim3 grid(p.GetGrid(N).first);

someCudaKernel<<<grid, threads>>>();
```

Для изменения значений данного параметра можно воспользоваться либо подсистемой оптимизациеи, которая подберёт наилучшее значение дл размера блока, либо методом `void SetNearestValue(size_t blockSize)`, который установит допустимое значение, наиболее близкое к требуемому.

Функциональность параметров

Большинство параметров обладают следующими общими методами, которые можно разделить на три категории:

- Построение деревьев параметров:
  - `void Attach(const char *name, ParameterGroup *group)`. Добавляет текущий параметр к требуемой группе под заданным именем. В один момент времени параметр может принадлежать только одной группе. В случае, если вместо группы передать нулевой указатель, то параметр будет добавлен в корень дерева пользовательских параметров.
  - `void Detach()`. «Отвязывает» текущий параметр из группы. В случае, если данный параметр использовался в оптимизационной сессии, то это приведёт к её «поломке».
  - `bool IsAttached()`. Возвращает `true`, если параметр принадлежит какой-либо группе, и `false` иначе.

```
ParameterGroup group;
Parameter<int> p1;
Parameter<float> p2;

p1.Attach(&group);                      //Attaching “p1” to “group”.
group.Attach(“My group”);               //Creating root-level group.
p2.Attach(“My parameter”);              //Creating root-level parameter.
//...
if (p1.IsAttached())
{
   p1.Detach();
   p1.Attach(“Another my parameter”);
}
```

- Обработка событий параметров:
  - `Event<ReleasableObject *> OnRelease()`. Событие, которое произойдёт после уничтожения параметра.
  - `Event<ValueChangedEventArgs<T> > &OnValueChanged()`. Событие, сигнализирующие об произошедшем изменении значения параметра.

```
void OnReleaseHandler(ReleasableObject *)
{ printf(“Parameter was destroyed\n”); }

void OnValueChangeHandler(ValueChangedEventArgs<T>)
{ printf(“Parameter's value was changed\n”); }

Parameter<int> p;
p.OnRelease() += MakeDelegate(OnReleaseHandler);
p.OnValueChanged() += MakeDelegate(OnValueChangeHandler);
p = 10;
p = 9;
p.OnValueChanged() -= MakeDelegate(OnValueChangeHandler);
p = 8;
```

- Изменение значений и поведения параметров
  - `const T &GetValue()`. Возвращает значение параметра. Практически полностью эквивалентно использованию оператора приведения, однако полезно для случаев, когда компилятор не может самостоятельно определить тип переменной. Примеры:

```
Parameter<int> p = 3;

printf("%d != 3", p);
printf("%d == 3", p.GetValue());
printf("%d == 3", (int)p);
```

- `SetValue(const T &value)`. Устанавливает новое значение параметра. Эквивалентен перегруженному оператору присваивания.
- `ReadOnlyMode GetReadOnlyMode()`. Возвращает режим работы параметра, определяемый одним из следующих флагов:
  - `None`. Значение параметра может быть изменено как самой программой, так и из внешних утилит.
  - `Utilities`. Запрещает изменять значение параметра из внешних утилит. Полезно для вывода значений внутренних переменных, изменение которых может привести к ошибке.
  - `Core`. Запрещает изменять значение параметра из основной программы.
  - `All`. Запрещает изменять значение параметра из программы и внешних утилит.
- `SetReadOnlyMode(ReadOnlyMode mode)`. Устанавливает новый режим работы параметра.

```
Parameter<std::string> currentAction = "None";
currentAction.SetReadOnlyMode(Utilities);
currentAction.Attach("Current action");

EnumParameter requiredAction("{ Reading, Writing, Processing }");
requiredAction = 0;
requiredAction.SetReadOnlyMode(Core);
requiredAction.Attach("Required action");

while (true)
{
   switch (requiredAction)
   {
      case 0:
         //...
         currentAction = "Reading data";
         break;
      case 1:
         //...
         currentAction = "Writing data";
         break;

      case 2:
         //...
         currentAction = "Processing data";
         break;
   }
}
```

Гибридные примитивы

Пространство имен ttgLib::Primitives

Для упрощения реализации различных паттернов параллельного программирования библиотека ttgLib предоставляет набор гибридных примитивов, позволяющих отобразить вычисления на все доступные вычислители. С помощью данных примитивов, являющихся обёртками поверх динамических параметров, можно сгруппировать несколько альтернативных вычислительных ядер, написанных с использованием технологий NVidia CUDA, OpenCL и/или расширений SSE/AVX, в результате чего при работе для каждого вычислителя будет выбрано оптимальное ядро, а вычислительная нагрузка равномерно распределена между всеми устройствами.

HybridFor

Данный примитив реализует паттерн parallel_for, позволяющий произвольный одномерный цикл заменить несколькими независимыми циклами, каждый из которых будет обрабатывать свою часть данных на целевом устройстве. Для этого необходимо создать экземпляр класса HybridFor, в котором требуется зарегистрировать доступные вычислительные ядра и который в дальнейшем будет использоваться для запуска вычислений. Примеры:

```
void cudaKernel(void *data, size_t lo, size_t hi)
{ /*Processing data array sub-range using current CUDA-device*/ }

void sseKernel(void *data, size_t lo, size_t hi)
{ /*Processing data array sub-range using SSE extensions*/ }

void sseOmpKernel(void *data, size_t lo, size_t hi)
{ /*Processing data array sub-range using SSE extensions and OpenMP directives*/ }

//...

HybridFor hFor;
hFor.CUDA() += cudaKernel;
hFor.Serial() += sseKernel;
hFor.Parallel() += sseOmpKernel;
std::vector<int> data;
//...
hFor.Process(data);
```

Детальное описание формата поддерживаемых ядер приведено в следующем разделе. Из специфичных особенностей данного примитива следует отметить следующие методы:

- `SetAlignment(size_t alignment)`. Позволяет задать выравнивание передаваемых в ядро интервалов, например, необходимое, для корректной работы расширений SSE.
- `SetMaxRange(size_t maxSize)`. *Under development*.

HybridTask **(not available in current version)**

Этот примитив предназначен для выполнения набора неделимых неоднородных задач, каждая из которых может быть выполнена на одном из доступных устройств, возможно, с использованием альтернативных вычислительных ядер. Это, например, позволяет равномерно загрузить все вычислители при выполнении пакетных заданий, причём для каждого из вычислителей будет подобрано не только оптимальное ядро, но и преимущественно ему будут передаваться те задачи, на которых он показывает максимальную производительность. Примеры:

```
void cudaKernel(void *data)
{ /*processing data using current CUDA-device*/ }

void cudaKernel_Textures(void *data)
{ /*processing data using current CUDA-device using texture memory*/ }

void cpuKernel(void *data)
{ /*processing data using current CUDA-device*/ }
```

```
//...

HybridTask hTask;
hTask.CUDA() += cudaKernel;
hTask.CUDA() += cudaKernel_Textures;
hTask.Serial() += cpuKernel;
std::vector<void *> data;
//...
hTask.Process(data);
```

Детальное описание формата поддерживаемых ядер приведено в следующем разделе. Из специфичных особенностей данного примитива следует отметить следующие методы:

– `SetMetrics(double (*f)(void *))`. Задаёт метрику для определения абстрактной сложности задачи (в произвольных единицах). Предполагается, что время выполнения задач на одном и том же ускорителе зависит линейно от их сложности, что позволяет более эффективно производить распределение нагрузки по всем вычислителям и подбирать оптимальные по размеру задачи для каждого типа устройства.

Формат ядер

Для задания вычислительных ядер требуется написать функцию или метод класса, которые на вход получают указатель на обрабатываемые данные, и, возможно, дополнительную информацию, после чего начинают вычисления. Аргументы подобных функций-ядер должны соответствовать одному из форматов, поддерживаемых примитивом:

Тип примитива	***Поддерживаемые форматы ядер***

`HybridFor`	`void kernel(void *data, size_t lo, size_t hi);`
	`void kernel(size_t lo, size_t hi);`
	`void kernel(HybridForKernelData data)`

`HybridTask`	`void kernel(void *data);`
	`void kernel();`
	`void kernel(HybridTaskKernelData data)`

Для случаев, когда ядро реализовано как функция, достаточно просто использовать перегруженный оператор `+=`. В случае же, если ядро реализовано статическим или обычным методом класса, то его требуется оформить в виде делегата с помощью функции `MakeDelegate()`. Примеры:

```
void functionKernel(HybridTaskKernelData data);

class MyClass
{
    public:
        void methodKernel(HybridTaskKernelData data);
        static void staticMethodKernel(HybridTaskKernelData data);
};

//...

HybridTask hTask;
hTask.Serial() += functionKernel;
hTask.Serial() += MakeDelegate(functionKernel);
hTask.Serial() += MakeDelegate(MyClass::staticMethodKernel);
MyClass mc;
hTask.Serial() += MakeDelegate(&mc, &MyClass::methodKernel);
```

Также иногда требуется передать в ядро набор динамических параметров, которые влияют на его производительность и которые имеет смысл оптимизировать. Для этого требуется создать группу параметров, содержащую все специфичные для данного ядра параметры, которую передать при регистрации с помощью вспомогательного класса. Примеры:

```
BoundedParameter<int> blockSize(10, 32);
ParameterGroup kernelParameters;

void kernel(void *data);

//...

HybridTask hTask;
blockSize.Attach(&kernelParameters);
hTask.Cuda() += HybridTaskKernel(kernel, &kernelParameters);
```

Переданные таким образом параметры будут независимо оптимизироваться с целью достижения максимальной производительности конкретного ядра. Стоит предупредить, что одно и то же ядро может независимо и одновременно выполняться несколькими устройствами (но с разными данными), однако это не вызовет проблем с доступом к параметрам ядра - для каждого устройства автоматически будет создана своя локальная копия параметров, оптимизация которой будет происходить независимо. Поэтому, например, в рассмотренном выше случае при обращении к параметру blockSize из ядра, выполняемого на CPU, может быть получено значение 16, а из ядра, запущенного на GPU – 32.

При работе каждое ядро будет выполняться в своём потоке, ассоциированным с целевым устройством. В случае произвольной ошибки ядро должно «выбросить» исключение типа `std::runtime_error` или его наследника, которое будет перехвачено и заново «выброшено» в основном потоке.

Для определения характеристик устройства, на котором выполняется данное ядро, в сервисе `DeviceManager` предусмотрены следующие методы:

- `CudaDevice *GetCurrentCudaDevice()`. Возвращает ссылку на объект, отвечающий текущему CUDA-устройству. В случае, если данный метод вызван вне примитива или метода `CudaDevice::Execute()`, то будет выброшено исключение.
- `X86Device *GetCurrentX86Device()`. Возвращает ссылку на объект, отвечающий текущему x86-совместимому устройству. В случае, если данный метод вызван вне примитива или метода `X86Device::Execute()`, то будет выброшено исключение.

Примеры:

```
void cudaKernel(void *data)
{
   CudaDevice *dev = DeviceManager::GetRef()->GetCurrentCudaDevice();
   double gflops;
   //...
   printf("%s performance: %lf Gflops\n", dev->GetStaticInfo()->GetName(),gflops);
}
```

Выбор устройств

Для регистрации ядер в примитивах используется разделение устройств по поддерживаемым ими API. Таким образом, если в системе установлен центральный процессор и графический ускоритель от NVidia, то библиотека ttgLib «обнаружит» в системе сразу четыре вычислителя — один x86-совместимый, два OpenCL-устройства и одно CUDA-устройство. В случае, если для одного физического устройства подходят сразу несколько вычислительных ядер, то при оптимизации будет выбрано наиболее оптимальное ядро.

При добавлении ядер в примитивы можно делать привязку к одному из следующих логических устройств:

- `CUDA()`. Отвечает графическому ускорителю, поддерживающему технологию NVidia CUDA.
- `Parallel()`. Позволяет задать вычислительное ядро, которое планирует задействовать

все ядра центрального процессора (например, с помощью технологии OpenMP).
- `Serial()`. Предназначен для вычислительных ядер, которые используют только одно ядро центрального процессора, что позволяет одновременно выполнять до N экземпляров данного ядра.

Стоит отметить, что в большинстве случаев использование последовательных ядер (`Serial`) оказывается эффективнее, чем параллельных (`Parallel`), т. к. в первом случае библиотека ttgLib может пожертвовать частью ресурсов CPU для повышения эффективности работы CUDA-драйвера, а также это позволяет лучше работать на процессорах, поддерживающих технологию HyperThreading за счёт отказа от виртуальных ресурсов. Другими словами, если оптимальным оказывается использование не всех видимых системе ядер центрального процессора, то при оптимизации библиотека ttgLib этим обязательно воспользуется.

Другой возможностью является задание ограничений поддерживаемых вычислительным ядром устройств. Для этого в соответствующий метод примитива необходимо передать делегат-селектор, возвращающий `true`, если ядро может быть выполнено на заданном устройстве, и `false` иначе. Примеры:

```
bool cudaSelector(CudaStaticInfo *info)       //we want only Fermi architecture
{ return info->GetMajor() >= 2; }

void cudaKernel(void *data);

//...

HybridTask hTask;
hTask.CUDA(MakeDelegate(cudaSelector)) += cudaKernel;
```

Также имеются следующие готовые селекторы:
- `X86Memory(size_t megabytes)`. Определяет требуемый ядру объём памяти на x86-совместимом устройстве (общий).
- `CudaCaps(size_t major, size_t minor)`. Определяет минимальные требования, предъявляемые ядром к архитектуре NVidia CUDA.
- `CudaMemory(size_t megabytes)`. Определяет требуемый ядру объём памяти на CUDA-ускорителе (общий).

Таким образом, вышеприведённый пример можно переписать в следующем виде:

```
void cudaKernel(void *data);

//...

HybridTask hTask;
hTask.CUDA(CudaCaps(2, 0)) += cudaKernel;
```

Подсистема оптимизации

Пространство имен ttgLib::Optimization

Ключевой подсистемой библиотеки ttgLib является механизм оптимизации, позволяющий для произвольной группы динамических параметров и/или гибридных примитивов подобрать оптимальные значения, обеспечивающие максимальную производительность. Для этого предполагается, что вычисления производятся итерационно, и на каждом шаге вычислительная сложность зависит линейно от размера обрабатываемых данных.

Схема работы

Для включения механизма оптимизации необходимо создать экземпляр класса `OptimizationSession`, в который передать группу с оптимизируемыми параметрами. После этого в начале каждой итерации необходимо вызвать метод `OptimizationSession::StartIteration()`, а в конце — метод `OptimizationSession::FinishIteration()`. Примеры:

```
BoundedParameter<int> blockSize(16, 128);
ParameterGroup group;
blockSize.Attach(group);

OptimizationSession session(&group);
while (true)
{
    session.StartIteration();
    //Computing using “blockSize”.
    session.FinishIteration();
}
```

В случае использования гибридных примитивов следует воспользоваться реализованным в них методом `GetOptimizationParameters()`, который возвращает все используемые в примитиве параметры. Примеры:

```
HybridFor hFor;
hFor.Serial() += serialKernel;
hFor.CUDA() += cudaKernel;

OptimizationSession session(hFor.GetOptimizationParameters());
while (true)
{
    session.StartIteration();
    hFor.Process(data);
    session.FinishIteration();
}
```

Стоит отметить, что в процессе оптимизации все используемые примитивы, группы параметров и просто параметры не должны уничтожаться или подключаться к другим группам. В противном случае оптимизационная сессия «сломается», и при попытке вызвать её методы будет «выброшено» исключение типа ttgLib::FatalError.

Оптимизационные сессии

В классе `OptimizationSession` реализованы следующие методы, предназначенные для управления процессом оптимизации:

- `void StartIteration()`. Информирует о начале новой итерации
- `void FinishIteration()`. Информирует о завершении итерации. Время между началом и концом итерации будет использовано как метрика для оптимизации общих параметров.
- `void FinishIteration(double time)`. Информирует о завершении итерации.

Переданное время будет использовано как метрика для оптимизации общих параметров.

- `void Suspend()`. «Замораживает» процесс оптимизации, в результате чего значения параметров не будут изменяться (однако статистика может продолжать собираться).
- `void Resume()`. Возобновляет процесс оптимизации. Текущие значения параметров не будут учитываться при дальнейшей оптимизации.
- `bool IsSuspended()`. Возвращает `true`, если процесс оптимизации «заморожен», и `false` иначе.
- `void Restart()`. Перезапускает процесс оптимизации, удаляя всю собранную статистику. Данный метод полезно вызывать при смене типа вычислений, или же при ручном добавлении/удалении оптимизационных параметров.
- `void Append(ParameterBase *p)`. Добавляет к текущей сессии ещё один параметр для оптимизации. Приводит к перезапуску процесса оптимизации.
- `void Append(ParameterGroup *g)`. Добавляет к текущей сессии все параметры из данной группы. Приводит к перезапуску процесса оптимизации.

Примеры:

```
OptimizationSession os;

EnumParameter ep("Left, Right, Center");
os.Append(&ep);

BoundedParameter<int> bp(10, 20);
os.Append(&bp)

HybridTask hTask;
hTask.Serial() += serialKernel;
os.Append(hTask.GetOptimizationParameters());

//...

int iterCount = 0;

while (true)
{
    os.StartIteration();

    //Computing.
    hTask.Process(data);
    ComputeSomethingUsingParameters(ep, bp);

    //Finishing statistics gathering and stopping optimization.
    if (iterCount == 100)
        os.Suspend();

    //Restarting optimization in order to ensure, that all is good.
    if (iterCount == 5000)
    {
        os.Restart();
        os.Resume();
        iterCount = 0;
    }

    os.FinishIteration();
    iterCount++;
}
```

Стратегии оптимизации

В большинстве случаев об оптимизируемом приложении известна дополнительная априорная информация — например, ожидаемое количество итераций, примерное время одной итерации, требуется ли время для «устаканивания» производительности и т.д.. Данную информацию можно использовать для уточнения стратегии оптимизации, тем самым повысив как её качество, так и

скорость обучения.
Для формализации стратегии оптимизации следует создать класс `OptimizationStrategy`, в котором выбрать подходящие настройки и алгоритмы. По умолчанию используется наиболее универсальная стратегия, рассчитанная на не менее 200 итераций, среднее время которых превышает 5 миллисекунд.

Общие параметры оптимизационных стратегий задаются с помощью следующих методов:

- `Parameters::Parameter<UInt32> &AggressiveIterationCount()`. Возвращает параметр, который определяет количество «агрессивных» итераций, в течение которых алгоритм имеет право существенно замедлять время работы программы в поисках глобального экстремума.
  Рекомендуемые значения — от 20 до 200 в зависимости от количества итераций и оптимизируемых параметров.
- `Parameters::Parameter<UInt32> &StabilizationIterationCount()`. Определяет количество итераций, которые необходимо «пропустить» перед началом оптимизации с целью стабилизации производительности. Крайне полезно для программ, совместно использующих CPU и несколько GPU.
  Рекомендуемые значения — от 0 до 10 в зависимости от количества используемых вычислителей.
- `Parameters::Parameter<Double> &IterationAggregation()`. Возвращает параметр, определяющий время в секундах для минимальной итерации. В случае, если реальное время окажется ниже требуемого, то реальные итерации будут группироваться в логические, время которых будет превышать заданный предел. Предназначено для оптимизации программ, время итераций в которых порядка 1-5 миллисекунд, т. к. позволяет нивелировать влияние накладных расходов и скачков производительности.
  Рекомендуемые значения — от 10 миллисекунд. При этом нужно учитывать, что при оптимизации будут использоваться именно логические итерации, поэтому слишком сильное укрупнение может привести к более долгому обучению.
- `Parameters::EnumParameter<> &BasicAlgorithm()`. Возвращает перечисление, определяющее используемый алгоритм для оптимизации произвольной группы параметров, не снабжённой дополнительной информацией. Поддерживаемые значения — `BasicAlgorithmType::Default`, `BasicAlgorithmType::Clicking` и `BasicAlgorithmType::Genetic`.
- `Parameters::EnumParameter<> &BalancingAlgorithm()`. Задаёт тип алгоритма, который будет применяться для оптимизации групп параметров, в которых имеется возможность проводить балансировку вычислительной нагрузки (в текущей версии — примитив `HybridFor`). Поддерживаемые значения – `BalancingAlgorithmType::Default`, `BalancingAlgorithmType::Sticky` и `BalancingAlgorithmType::LUT`.

Для использования требуемой стратегии оптимизации её требуется передать в конструктор оптимизационной сессии. Примеры:

```
OptimizationStrategy strategy;
strategy.AggressiveIterationCount() = 100;
strategy.IterationAggregation() = 0.01;
strategy.BasicAlgorithm() = BasicAlgorithmType::Clicking;
strategy.BalancingAlgorithm() = BalancingAlgorithmType::LUT;

OptimizationSession os(strategy);

//...
```

Базовые алгоритмы оптимизации

Данные алгоритмы предназначены для оптимизации произвольных параметров, априорная информация о которых неизвестна. К примеру, для любых параметров, переданных для оптимизации вне рамок гибридных примитивов, будут использованы алгоритмы из данного класса.

На настоящий момент реализовано два следующих алгоритма данного типа:

- `Clicking`. Является модифицированной версией покоординатного спуска. В

«агрессивном» режиме перебирает все комбинации параметров в порядке убывания их производительности, а в основном — пытается удержаться в найденном экстремуме.
Данный алгоритм предназначен для случаев, когда имеется небольшое количество оптимизируемых параметров (порядка 1-4), или же число «агрессивных» итераций не превышает 200-500, так как обеспечивает лучшую скорость обучения и требует минимальное количество замеров.
В стратегии оптимизации можно задать следующие настройки данного алгоритма (через метод `ClickingAlgorithmSettings()`):

- `Parameters::BoundedParameter<double> &Quality()`. Нормированный параметр, принимающий значения от 0.0 до 1.0 и отвечающий за качество оптимизации. Чем выше качество, тем ниже скорость обучения, и наоборот.

– `Genetic`. Является реализацией генетического алгоритма. В «агрессивном» режиме проводит активные мутации и скрещивания лучших наборов параметров, в обычном — выбирает лучше наборы из оставшейся популяции.
Алгоритм предназначен для длительных вычислений с количеством «агрессивных» итераций, превышающим 1000, и не имеющих «скачков» в производительности. Также полезен при оптимизации очень больших наборов параметров (более 10).
В стратегии оптимизации можно задать следующие настройки данного алгоритма (через метод `GeneticAlgorithmSettings()`):

- `Parameters::BoundedParameter<double> &Quality()`. Нормированный параметр, принимающий значения от 0.0 до 1.0 и отвечающий за качество оптимизации (размер популяции и количество мутаций). Чем выше качество, тем ниже скорость обучения, и наоборот.

Примеры:

```
OptimizationStrategy strategy;
strategy.BasicAlgorithm() = BasicAlgorithmType::Genetic;
strategy.GeneticAlgorithmSettings().Quality() = 0.42;

OptimizationSession os(strategy);

//...
```

Алгоритмы балансировки нагрузки

Алгоритмы данного типа предназначены для балансировки вычислительной нагрузки между различными устройствами. В частности, примитив `HybridFor` будет оптимизироваться именно с помощью подобных алгоритмов.

На настоящий момент реализовано два следующих алгоритма данного типа:

– `Sticky`. Основной алгоритм, который распределяет нагрузку и выбор вычислительных ядер в зависимости от текущей производительности каждого устройства. Имеет два порога, первый из которых определяет минимальную порцию данных, которую имеет смысл отправлять на отдельный вычислитель, и второй - максимальную порцию, при превышении которой задача будет эксклюзивно обрабатываться одним устройством.
Предназначен для случаев, когда имеется несколько вычислителей со схожими архитектурными особенностями и/или нет времени на проведение дополнительного обучения.
В стратегии оптимизации можно задать следующие настройки данного алгоритма (через метод `StickyAlgorithmSettings()`):

- `Parameters::BoundedParameter<double> &ExcludeThreshold()`. Нормированный параметр, принимающий значения от 0.0 до 1.0 и определяющий нижний порог «прилипания». Например, если порог равен 0.2 и одному устройству должно быть передано 15% от всех данных, то целевое устройство использоваться не будет.
Рекомендуемое значение — 0.1.
- `Parameters::BoundedParameter<double> &ExclusiveThreshold()`. Нормированный параметр, принимающий значения от 0.0 до 1.0 и определяющий верхний порог «прилипания». К примеру, если порог равен 0.8, и одному устройству должно быть передано 90% от всех данных, то все вычисления будут проводиться только на целевом устройстве.

Рекомендуемое значение — 0.9.

– `LUT`. При работе данный алгоритм строит функции производительности каждого отдельного устройства в зависимости от размера обрабатываемых данных, что позволяет учитывать и использовать кэш-эффекты и прочие резкие «скачки» производительности.
Предназначен для случаев, когда используются вычислители с разной архитектурой и/или имеется достаточное количество альтернативных вычислительных ядер (более 2 веток на одно устройство).
В стратегии оптимизации можно задать следующие настройки данного алгоритма (через метод `GetLutAlgorithmSettings()`):
  – `Parameters::BoundedParameter<double> &ExcludeThreshold()`. Нормированный параметр, принимающий значения от 0.0 до 1.0 и определяющий нижний порог «прилипания». Например, если порог равен 0.2 и одному устройству должно быть передано 15% от всех данных, то целевое устройство использоваться не будет.
  Рекомендуемое значение — 0.1.
  – `Parameters::BoundedParameter<UInt32> &Granularity()`. Параметр, принимающий значения от 1 до 100 и определяющий количество точек, по которым строится дискретная функция производительности.
  Рекомендуемое значение — 10. Большее количество точек требует большее количество агрессивных итераций.

Примеры:

```
OptimizationStrategy strategy;
strategy.BalancingAlgorithm() = BalancingAlgorithmType::LUT;
strategy.LutAlgorithmSettings().ExcludeThreshold() = 0.1;
strategy.LutAlgorithmSettings().Granularity() =  5;

OptimizationSession os(strategy);

//...
```