Федеральное государственное образовательное учреждение

Высшего профессионального образования

Санкт-Петербургский государственный университет

Физический факультет

Кафедра вычислительной физики

Соснов Дмитрий Евгеньевич

Реализация алгоритмов реконструкции откликов частиц в TPC детектора NICA/MPD на гибридных вычислительных системах

БАКАЛАВРСКАЯ РАБОТА

Научный руководитель

к.ф.-м.н., доцент, кафедра вычислительной физики

Физический факультет СПбГУ _________________ С. А. Немнюгин

Научный руководитель

к.ф.-м.н, начальник сектора,

лаборатория физики высоких энергий,

НЭОМД, ЛФВЭ _________________ О. В. Рогачевский

Рецензент

м.н.с., НЭОМД, ЛФВЭ _________________ С. П. Мерц

Санкт-Петербург

2013

Содержание

Введение **2**

1 Описание ускорительного комплекса NICA **4**

1.1 Устройство детектора MPD 4

1.2 Строение камеры TPC 4

1.3 Схема работы камеры TPC 5

2 Описание вычислительной системы и программной модели NVIDIA CUDA **8**

2.1 История развития графических адаптеров 8

2.2 Модель программирования NVIDIA CUDA 8

2.3 Модель памяти NVIDIA CUDA 9

3 Применение NVIDIA CUDA для ускорения работы класса TpcDigitizerTask **11**

3.1 Входные данные 11

3.2 Последовательный алгоритм 11

3.3 Параллельный алгоритм 11

3.4 Реализация параллельного алгоритма 12

3.5 Изменение в системах Root и MpdRoot 16

3.6 Апробация алгоритма 16

Выводы **18**

Используемая литература **19**

A Исходный код файла TpcDigitizerTask.h **21**

B Исходный код файла TpcDigitizerTask.cu **26**

C Исходный код файла cuda_TMath.patch **59**

D Исходный код файла cuda_CMake.patch **60**

Введение

Изучение экстремально плотной и горячей ядерной материи является актуальной задачей современной физики. Особый интерес связан с изучением нового состояния материи, позволяющим пролить свет на наиболее фундаментальные проблемы физики - кварк-глюонной плазмы (КГП), существование которой было предсказано современной теорией сильного взаимодействия [1]. Особое внимание уделялось изучению свойств КГП при энергиях столкновений 20-100 ГэВ на нуклон [2] [3].

Экспериментально доказано существование нового состояния вещества [1]. Вместе с тем недостаточно исследованы свойства КГП при энергиях от 2 до 10 ГэВ. Для получения подобных экспериментальных данных проектируются и строятся несколько ускорителей, в том числе и ускорительный комплекс NICA (Nuclotron-based Ion Collider fAcility) на базе Нуклотрона, рассчитанный на изучение столкновений ионов с энергиями до 11ГэВ и располагающийся в Объединенном институте ядерных исследований (ОИЯИ) в г. Дубна[4]. Схема ускорительного комплекса представлена на рисунке 1.

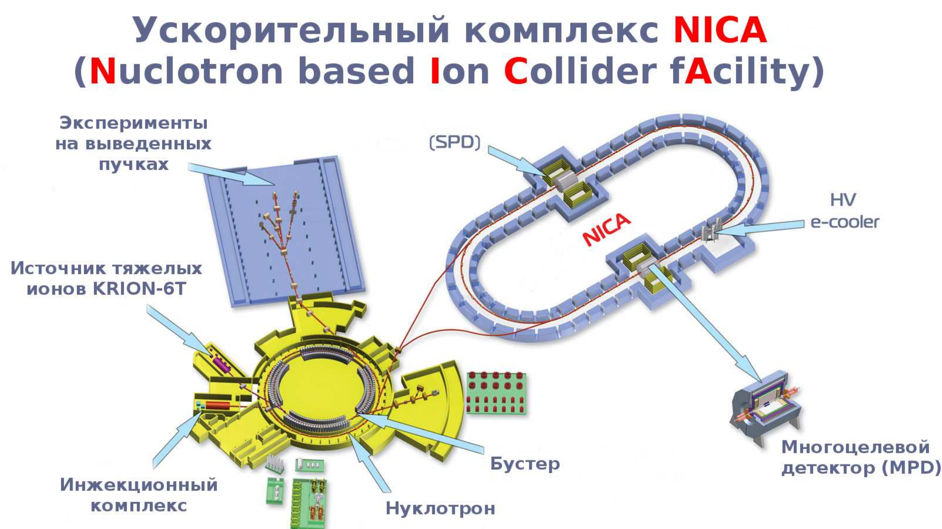

Рис. 1 – Схема ускорительного комплекса NICA

На коллайдере предусмотрена возможность размещения двух детекторов для проведения двух экспериментов одновременно. Один из детекторов - Многоцелевой детектор (Multi-Purpose Detector, MPD) планируется для изучения свойств горячей и плотной ядерной материи, образованной при соударении тяжёлых ионов высоких энергий [4].

Для компьютерной обработки экспериментальных результатов с помощью программных пакетов Root [5] и FairRoot [6], разрабатываемых в научных центрах CERN [8] и GSI (FAIR) [9], в ОИЯИ ведётся разработка программного комплекса MpdRoot [7]. Для апробации алгоритмов компьютерной обработки экспериментальных данных необходимо проведение моделирования столкновения в детекторе, также производимое при помощи системы MpdRoot.

Одним из важных параметров при проведении моделирования для отработки алгоритмов последующего расчёта является скорость выполнения, потому для уменьшения времени моделирования возможно применять технологии, использующие дополнительные вычислительные мощности, отличные от процессора. Одной из таких технологий является архитектура параллельных вычислений NVIDA CUDA, позволяющая использовать мощности графических процессоров общего назначения (GPGPU).

Данная работа посвящена оптимизации моделирования событий, происходящих во время-проекционной камере (TPC), с применением технологии NVIDIA CUDA. Целью работы являлась адаптация производящего моделирование временно-проекционной камеры класса программного комплекса MpdRoot под использование технологии NVIDIA CUDA. Актуальность представленной работы обосновывается необходимостью уменьшения затрачиваемого на моделирование времени для более эффективной отработки алгоритмов обработки данных, принимаемых с многоцелевого детектора, и необходимость создания методов использования гибридных вычислительных систем для увеличения производительности вычислений систем программного комплекса MpdRoot.

1 Описание ускорительного комплекса NICA

1.1 Устройство детектора MPD

На коллайдере NICA предусмотрена возможность установки двух детекторов для одновременного проведения нескольких экспериментов. На текущий момент запланированным детектором является многоцелевой детектор MPD [10]. Схема многоцелевого детектора представлена на рисунке 2.

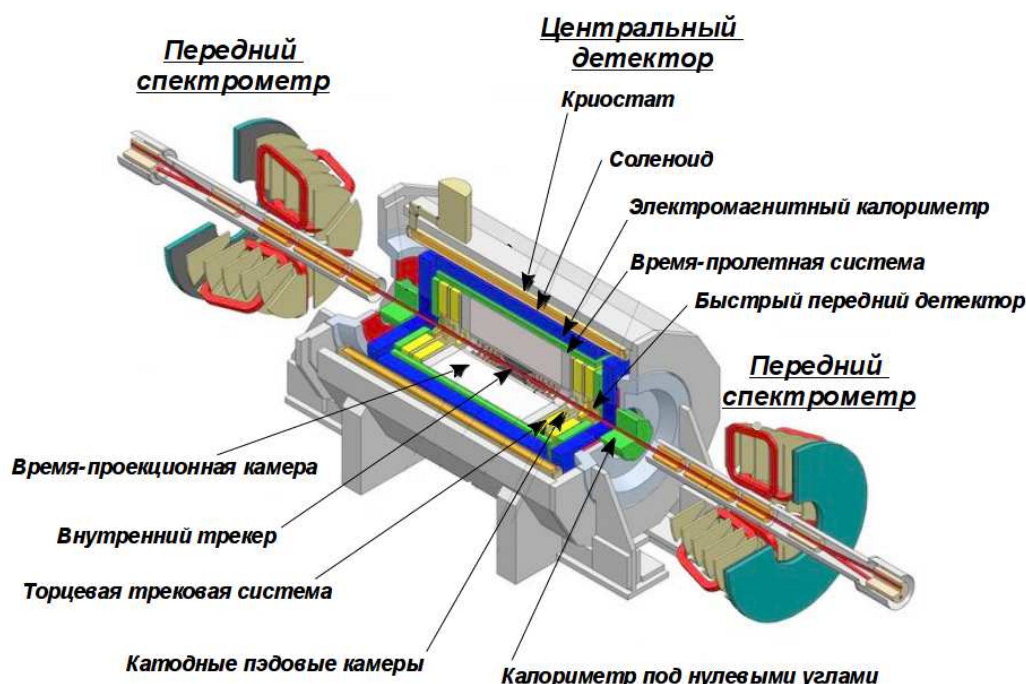

Рис. 2 – Схема многоцелевого детектора

1.2 Строение камеры TPC

Время-проекционная камера является основным трековым детектором MPD. Совместно с внутренней трековой системой, системой времени пролёта и электронным калориметром, TPC обеспечивает точное измерение импульса заряженных частиц и их идентификацию [4].

Камера представляет собой двенадцатигранный цилиндр с внутренним радиусом в 35 см. и внешним в 110 см. Посредине цилиндра располагается высоковольтный электрод, служащий для удаления положительных ионов из камеры, а сама камера наполнена смесью из аргона (90%) и метана (10%). В торцевых стенках камеры находятся считывающие

плоскости, разделенные по секторам. Каждый сектор считывающих плоскостей имеет в себе считывающие пластины (пэды) различных размеров для внутренней и внешней области. Перед считывающими плоскостями находится система, служащая для усиления попадающего на считывающие пластины заряда (пропорциональная камера). Структура время-проекционной камеры представлена на рисунке 3. Характеристики камеры представлены в таблице 1.

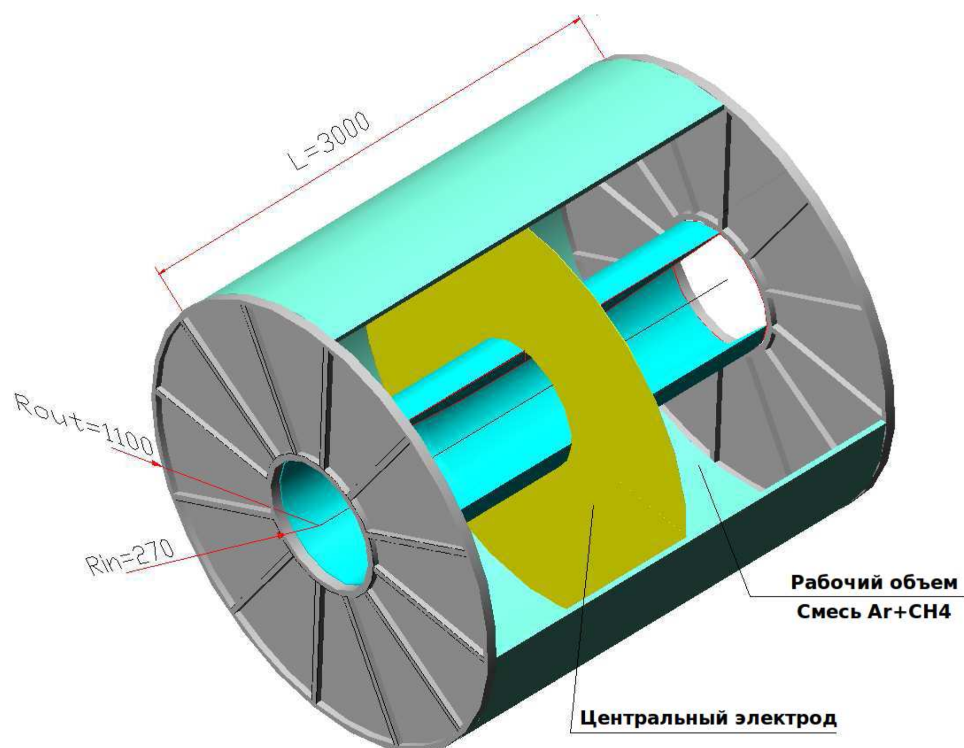

Рис. 3 – Схема время-проекционной камеры

1.3 Схема работы камеры TPC

После столкновения во внутреннем трекере получившиеся частицы пролетают дрейфовый объем камеры и ионизируют находящийся в ней газ. Положительно заряженные ионы начинают движение по направлению к центральному электроду, электроны - к анодной сетке пропорциональной камеры. Падающий на анод электрон выбивает лавину вторичных электронов по направлению к считывающей плоскости, в то время как запирающая сетка предотвращает попадание электронов вторичной лавины в дрейфовый объем камеры. Электроны вторичной плоскости передают заряд на пэды. Схема работы камеры представлена на рисунке 4 для одной из половин камеры.

Внешний радиус, см	110
Внутренний радиус, см	35
Количество секторов камеры	12 (в каждой стороне)
Количество считывающих плоскостей	12 (в каждой стороне)
Ширина внутренней области пэдов, шт.	21
Ширина пэда внутренней области, мм	4
Высота пэда внутренней области, мм	12
Ширина внешней области пэдов, шт.	30
Ширина пэда внешней области, мм	5
Высота пэда внешней области, мм	18

Таблица 1 – Параметры время-проекционной камеры

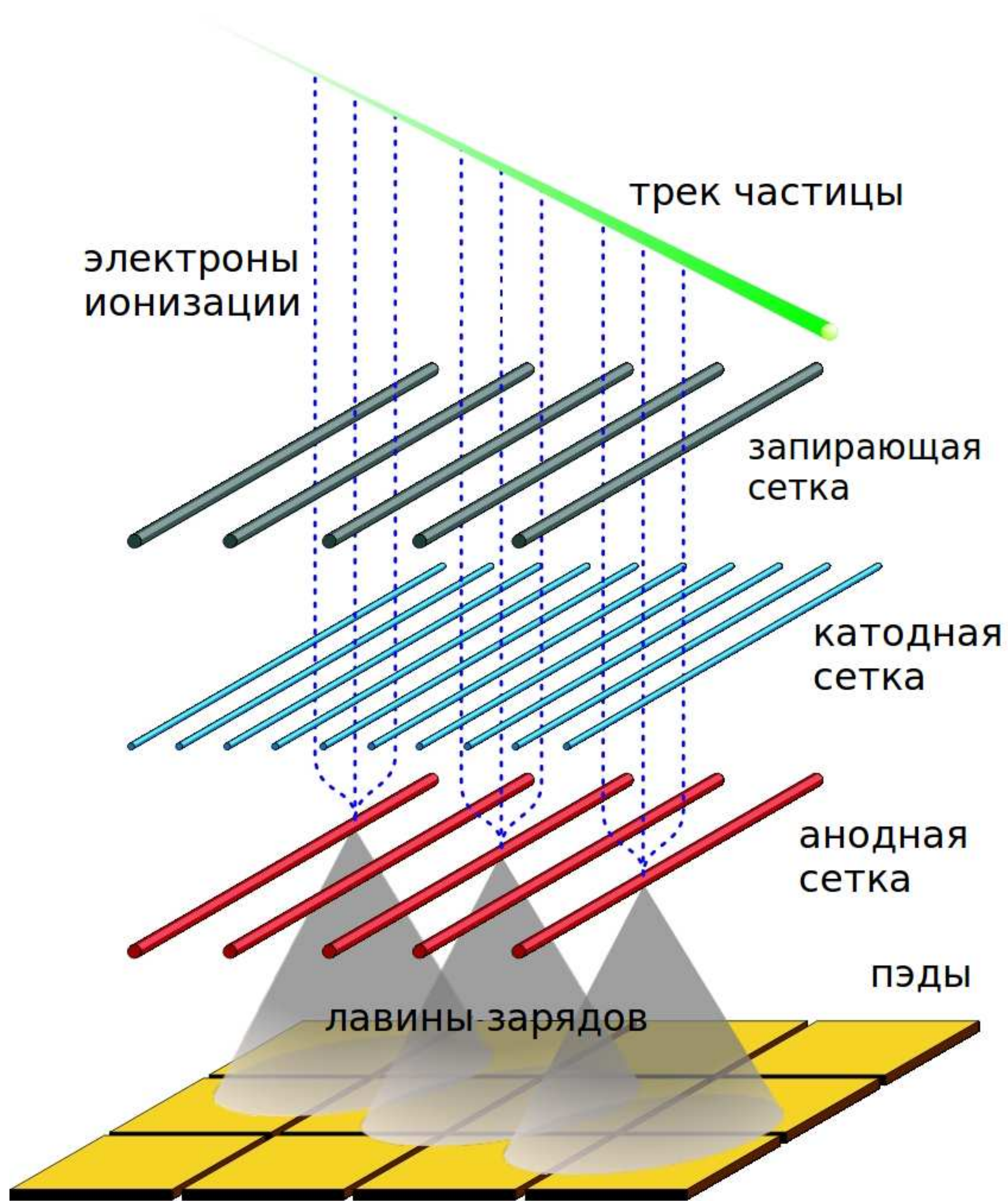

Рис. 4 – Схема работы время-проекционной камеры

2 Описание вычислительной системы и программной модели NVIDIA CUDA

2.1 История развития графических адаптеров

Первым адаптером является выпущенный в 1981 году текстовый адаптер IBM MDA (Monochrome Display Adapter), функция которого заключалась в выводе содержимого видеопамяти на экран. Адаптером поддерживался вывод 25 строк по 80 символов в каждой. Суммарное разрешение было 720x350 пикселей. Первым же графическим адаптером являлся вышедший в том же году адаптер IBM CGA (Color Graphics Adapter), поддерживающий графический режим вывода помимо текстового, при этом все вычисления над пикселами производились процессором. Вплоть до 1996 года усложнение видеоадаптеров велось путем увеличения количества рабочих пикселов.

В 1996 году был выпущен адаптер 3dfx Voodoo Graphics, который обладал кроме видеопамяти ещё и процессором, служащим для вычисления работы с текстурами. В 1998 году был выпущен адаптер с архитектурой Voodoo2, который позволял, благодаря наличию двух текстурных блоков, накладывать до двух текстур за проход. В дальнейшем улучшение графических адаптеров шло путём усложнения вычислительных блоков. Также, поскольку вычисления над отдельными пикселами независимы, архитектуры пришли к большому числу вычислительных устройств работающих в параллельном режиме [12].

В ноябре 2006 компанией NVIDIA была разработана технология NVIDA CUDA, позволяющая использовать аппаратные ресурсы графических адаптеров для пользовательских вычислений. Благодаря направленности видеокарт на работу с независимыми пикселами, использование технологии CUDA подразумевает использование параллельных вычислений [11].

2.2 Модель программирования NVIDIA CUDA

Модель программирования CUDA [11] расширяет язык программирования C++, позволяя пользователю определить функции, называемые ядрами, которые при вызове выполняются параллельно указанное количество раз на различных потоках, каждая в отдельном потоке, в отличие от однопоточного исполнения обычных функций языка C++. Также необходимо учитывать ограничение применения ядер CUDA к структуре классов C++: для предотвращения запуска ядра отдельным потоком CUDA-ядро не может

являться методом класса. Каждому потоку, выполняемому параллельно, присваивается уникальный идентификатор, который доступен через встроенную трехкомпонентную переменную, поэтому потоки могут объединяться в одномерные, двумерные и трехмерные блоки нитей (thread block). Так как потоки каждого блока одновременно используют одни и те же ограниченные ресурсы памяти и вычислительные ресурсы, максимальные размеры блока ограничены. Для увеличения производительности блоки, в свою очередь, объединяются в одномерную, двумерную или трехмерную сетку блоков (grid), потому максимальная величина исполняемых потоков равна произведению числа потоков в блоке нитей на количество блоков в сетке. Вычисления блоков в сетке должны быть абсолютно независимыми, в отличие от работы потоков внутри блока, которая может быть скоординирована путём использования функции барьерной синхронизации __*syncthreads()*. Распределение потоков по сетке и бокам определяется специальной структурой *<<<Db, Dt>>>* в вызове ядра между именем и принимаемыми параметрами, где первый элемент отвечает за распределение потоков в блоке, а второй за распределение блоков в сетке блоков [11].

Также важной единицей является размер “варпа” - количество потоков выполняемых физически одновременно. Для повышения производительности количество потоков, задействованных в ядре, следует использовать кратное половине “варпа”, так как некоторые виды запросов обращений к памяти могут быть объединены в один запрос, что существенно повышает скорость работы с памятью [11].

Приведённая выше технология подразумевает использование пульсирующей модели(Fork/Join) модели исполнения, заключающейся в исполнении участков с последовательным алгоритмом между вызовами ядер CUDA, выполняемых в параллельном режиме.

2.3 Модель памяти NVIDIA CUDA

Для эффективного использования ресурсов память устройства разделена на элементы с различной латентностью и параметрами доступа. В наличии имеется локальная (регистровая) память (local memory), собственная у каждого потока и имеющая время жизни равное времени жизни потока, распределенная память (shared memory), единая для всех потоков одного блока, и несколько видов памяти общего доступа - глобальная, константная и текстурная. Если две первых упомянутых памяти имеют малую латентность (время доступа к памяти), то память с общим доступом имеет большую латентность, при этом имея кэширование константной памяти для любых устройств с поддержкой технологии CUDA и кэширование памяти общего назначения для устройств с параметром вычислительной возможности (compute capability) не менее 2.0. Параметр вычислительной возможности определяет архитектуру CUDA-устройства, главный и

минорный номера, определяющие глобальную архитектуру устройства и номер последовательных улучшений архитектуры (возможно, включающих в себя новые возможности) соответственно [11]. Структура доступа к памяти в технологии NVIDIA CUDA представлена на рисунке 5.

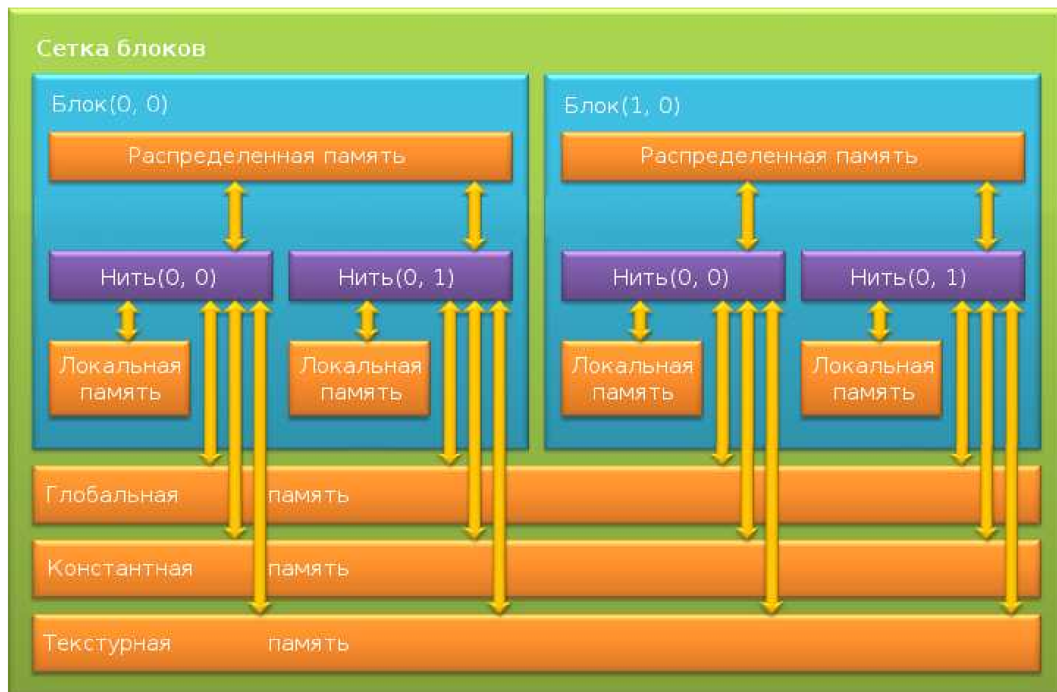

Рис. 5 – Структура памяти в технологии NVIDIA CUDA

Именно благодаря ограниченности размеров и подобной структуре памяти при работе с технологией CUDA необходимо уделять особое внимание работе с системой памяти.

3 Применение NVIDIA CUDA для ускорения работы класса TpcDigitizerTask

3.1 Входные данные

Входными данными для класса являются данные, полученные с выбранного генератора событий и обработанные системой моделирования взаимодействия пучков частиц Geant 4 [13], представленные в виде привязанных к определённому треку частицы наборов координат точек, в которых измерена энергия частицы.

3.2 Последовательный алгоритм

В последовательном алгоритме вычисления проводятся следующим образом: для каждой точки замера энергии вычисляется потеря энергии с момента предыдущего замера. Считая, что вся потерянная энергия считается потраченной на ионизацию газа, вычисляется количество электронов и ионов, полученных при ионизации, и распределяется между двумя точками замера. В последовательном алгоритме для каждого из ионов производится сдвиг к центральному электроду, а для электронов - к пропорциональной камере. При необходимости дошедшие до центрального электрода ионы удаляются, заряд электронов, прошедших дрейфовый объем и дошедших до пропорциональной камеры увеличивается благодаря электронам вторичной лавины и считывается, попадая на считывающие плоскости. Для увеличения производительности при расчёте воздействия вторичной лавины на считывающие пластины вводится ограничение на размер области воздействия электронов вторичной лавины со считывающей плоскостью - “радиус обрезания”. Использование подобного ограничения возможно ввиду нормального по координатам считывающей плоскости распределения электронов во вторичной лавине.

Для дальнейшей обработки полученные значения зарядов на считывающих плоскостях собираются в единый выходной четырёхмерный массив. Блок-схема последовательного алгоритма представлена на рисунке 6.

3.3 Параллельный алгоритм

В параллельном алгоритме определение количества и параметров ионизированных электронов происходит аналогично последовательному алгоритму, а моделирование прохождения дрейфового объёма и усиления в пропорциональной камере происходит в параллельном режиме. Так как вклад в выходные данные от каждого ионизованного электрона равноценен, к полной задаче не могут применяться алгоритмы, основанные на параллелизме по данным всей задачи. В то же

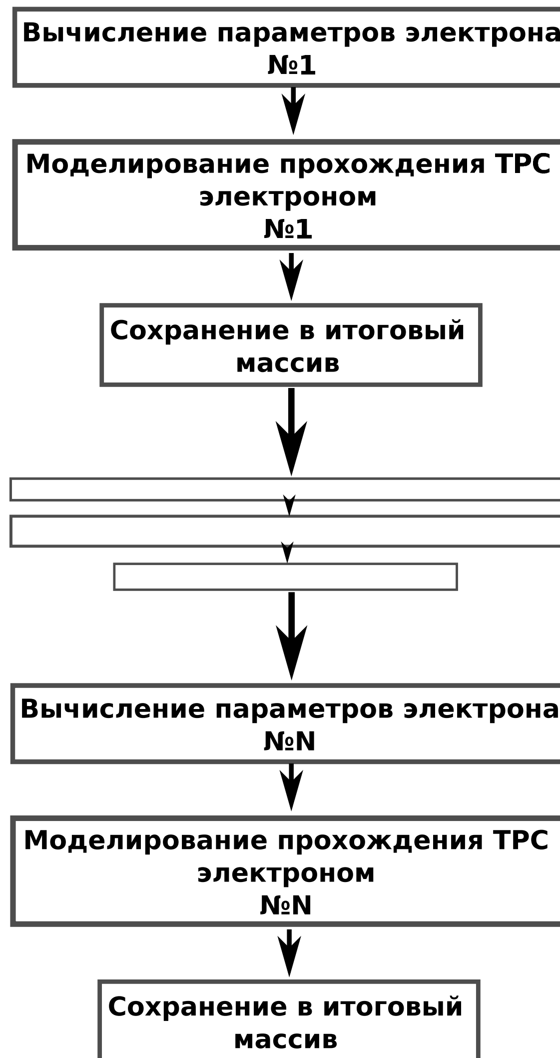

Рис. 6 – Блок-схема последовательного алгоритма

время, моделирование прохождения камеры каждым электроном и нахождение заряда, переданного вторичной лавиной может выполняться абсолютно независимо и для этой части вычислений применение параллельных алгоритмов должно давать ощутимый вклад в скорость вычисления. Именно поэтому было решено выполнить реализацию параллельного алгоритма с использованием параллельной системы вычислений NVIDA CUDA. Блок-схема параллельного алгоритма представлена на рисунке 7

3.4 Реализация параллельного алгоритма

При реализации вышеописанного алгоритма необходимо учитывать сложную систему памяти, используемую в технологии CUDA и, сравнительно с объёмом оперативной памяти, малый объем доступной видеопамяти. С учётом этого, для выполнения параллельной части вычисле-

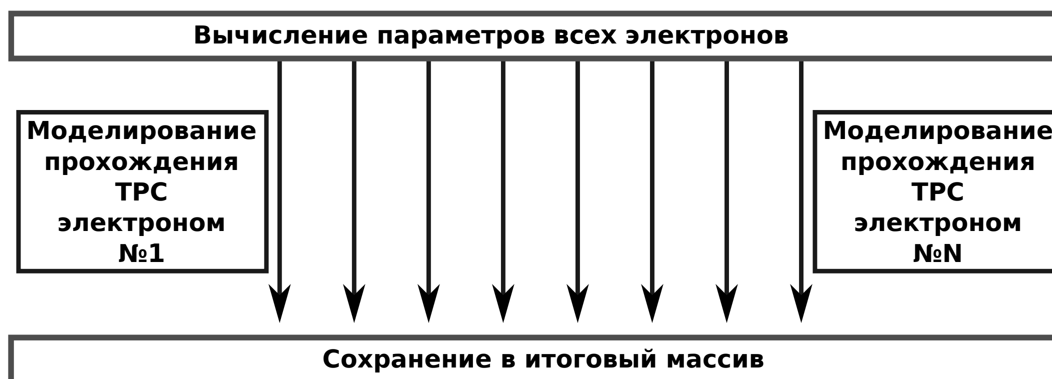

Рис. 7 – Блок-схема параллельного алгоритма

ний используется вариант запуска ядра *driftKernelCuda* с одномерной сеткой одномерных блоков, упрощающий масштабирование программы на доступные ресурсы. Поскольку необходима поддержка моделирования как в параллельном, так и в последовательном режиме с возможностью компиляции исходного кода компиляторами без поддержки технологии NVIDIA CUDA, спецификаторы __*global*__, __*device*__ и __*host*__, определяющие целевое устройство выполнения функции, определяются при помощи препроцессорной директивы макроподстановки *#define* при использовании компиляторов без поддержки CUDA. Таким же образом экранируются спецификаторы памяти __*constant*__ и __*shared*__ и вызовы CUDA-ядер.

Ввиду малого размера локальной и регистровой памяти и большого времени доступа к глобальной памяти, в глобальной памяти CUDA-устройства содержатся только редко используемые входной и выходные массивы большого размера. Также для оптимизации доступа глобальные переменные класса были перенесены в константную память. Подобное перенесение возможно по причине того, что одной и той же переменной-указателю на константную память CUDA-устройства для каждого устройства отвечает свой указатель на конкретную константную память устройства. Также ввиду ограниченного размера локальной памяти потока некоторые часто используемые переменные большого размера, такие как текущие параметры электрона, были расположены в распределенной памяти блока потоков.

Для исполнения кода одновременно на нескольких устройствах в начале функции, распределяющей задания на CUDA-устройства *TpcDigitizerTask:: DistributorCuda* определяются параметры вычислительной возможности доступных устройств, составляются список из устройств, отвечающих минимальным требованиям, заданным переменной *minimal_capability*, и выполняет распределение задач на отобранные устройства. При отсутствии подходящих устройств вычисления выполняются для каждого электрона в последовательном варианте. Полный листинг полученного класса представлен в приложениях A, B.

Таким образом, параллельная реализация вычислений выполнена следующим образом:

- Система MpdRoot подключает класс *TpcDigitizerTask* и инициализирует параметры по умолчанию, используя наследуемую от базового класса *FairTask* функцию *TpcDigitizerTask:: Init*, необходимую для корректного взаимодействия между классами системы *MpdRoot*.

- Для выполнения моделирования *MpdRoot* вызывает используя наследуемую от базового класса *FairTask* функцию *TpcDigitizerTask:: Exec*, которая определяет количество электронов, полученных при ионизации газа и, при компиляции компилятором с поддержкой технологии NVIDIA CUDA, вызывает функцию распределения электронов между вычислительными ресурсами *TpcDigitizerTask:: DistributorCuda*. При компиляции не поддерживающим технологию CUDA компилятором для дальнейших вычислений вызывается функция вычисления с использованием центрального процессора *TpcDigitizerTask:: DistributorCPU*

- В случае выполнения метода *TpcDigitizerTask:: DistributorCuda* в нем производится определение параметров доступных вычислительных ресурсов GPGPU. В случае отсутствия приемлемых CUDA-ресурсов выполняется передача управления функции *TpcDigitizerTask:: DistributorCPU*, иначе создаётся вектор (объект класса *vector* стандартной библиотеки STL [14]) из подходящих устройств. Для каждого из устройств определяется размер "варпа", на основе которого для каждого устройства вычисляется максимальное количество потоков, используемое при запуске ядра и число потоков в блоке. Для возможности корректного выполнения на нескольких устройствах для каждого устройства создаётся отдельный логически независимый поток вычисления. Для каждого из полученных потоков выполняется распределение общей памяти, заполнение константной памяти и вызов ядра *setupCudaRandom*, обеспечивающего создание независимых псевдослучайных последовательностей для каждого потока [15].

- Для каждого потока выполняется определение количество электронов из необработанной части входного массива для расчёта на устройстве в текущей итерации и загрузка данных в общую память устройства. После загрузки параметров электронов для каждого потока выполняется запуск ядра *driftKernelCuda*, находящегося вне класса ввиду ограничений технологии CUDA, описанных в главе 2.2.

- В вычислительном ядре *driftKernelCuda*, скрытом от компиляторов, не поддерживающих технологию NVIDIA CUDA, для каждого электрона в отдельной нити исполнения моделируется прохождение дрейфового объёма камеры, при этом диффузия моделируется введением трехмерного вектора диффузии *diffuse* с элементами, равными случайной величине

с нормальным по координате распределением, отвечающими величине вызванного диффузией смещения электрона. По данным прохождения электронов дрейфового объёма вычисляется время, сектор появления лавины вторичных электронов на анодной плоскости, а также локальные секторные координаты схода лавины. Для определения затронутых электронами лавины считывающих пластин и полученного ими заряда вызывается функция *TpcDigitizerTask:: GetAreaCuda*.

- Функция *TpcDigitizerTask:: GetAreaCuda* по полученным координатам появления вторичной лавины последовательно находит все пэды, затронутые лавиной и вызывает для них функцию *TpcDigitizerTask:: CalculatePadResponseCuda* для определения полученного заряда.

- В функции *TpcDigitizerTask:: CalculatePadResponseCuda* величина полученного заряда определяется исходя из полученного при создании класса коэффициента усиления вторичной лавины и “радиуса обрезания”.

- После завершения выполнения ядра управление передается функции *TpcDigitizerTask:: DistributorCuda*, которая объединяет полученные данные в четырехмерный массив *fDigitsArray*, являющийся объектом класса и отвечающий за заряд на пэдах. Функция *TpcDigitizerTask:: DistributorCuda* повторяет вычисления для тех электронов из входного массива, для которых не проводилось моделирование.

- При компиляции компилятором без поддержки технологии NVIDIA CUDA или при отсутствии требуемых вычислительных устройств вычисление производится функциями *TpcDigitizerTask:: DistributorCPU*, *TpcDigitizerTask:: GetAreaCPU* и *TpcDigitizerTask:: CalculatePadResponseCPU*: функция *DistributorCPU* в последовательном режиме выполняет моделирование прохождение дрейфового объёма аналогично функции *driftKernelCuda* и вызывает функцию *TpcDigitizerTask:: GetAreaCPU* для определения пэдов, подвергшихся воздействию вторичной лавины электронов. Отличием в функциях последовательного вычисления от функций вычисления на CUDA-устройствах является использование переменных класса, в отличие от переменных, размещённых в константной памяти CUDA-устройств, и использование переменных типа *std::vector* вместо массивов с количеством элементов, равным максимальному количеству возможных засвеченных считывающих элементов.

- Дальнейшее выполнение производится в функции *TpcDigitizerTask:: Exec*, выполняющей сохранение полученных данных в структуре *store->fBuffer*, являющейся выходными данными для класса *TpcDigitizerTask*.

3.5 Изменение в системах Root и MpdRoot

Для компиляции с использованием компилятора NVCC от компании NVIDIA, поддерживающего технологию NVIDIA CUDA для подключения библиотеки *TMath.h* системы Root необходимо изменение файла *TMath.h* в области определения функции *TMath::IsNaN*. Для корректного изменения у всех пользователей системы MpdRoot был создан файл правки *cuda_ TMath.patch*, листинг которого содержится в приложении C.

Для компиляции отдельных файлов с использованием компилятора NVCC был изменен файл управления сборкой *tpc/CMakeLists.txt* для системы автоматизации сборки CMake. Листинг изменений, записанный в файл *cuda_ CMake.patch*, представлен в приложении D.

3.6 Апробация алгоритма

Апробация приведённого алгоритма проводилась на данных, полученных генератором событий URQMD (The Ultrarelativistic Quantum Molecular Dynamics) [16] на основе данных столкновения двух ядер золота при энергии 9 ГэВ. Вычисления производились на персональном компьютере с процессором *Intel Core i5-3210M* и графическим вычислительным устройством *NVIDIA GeForce GT 630M*, имеющем в себе 96 CUDA ядер, под управлением операционной системы *ArchLinux* с ядром *3.9.4-1*. Полученные времена выполнения вычислений приведены в табл. 2. Как видно из таблицы, полученные алгоритмы дают значительный прирост в производительности моделирования камеры TPC, получая более чем десятикратное ускорение даже на графическим устройстве с низкими характеристиками.

Номер запуска	Старая последовательная реализация, *мс*	Новая последовательная реализация, *мс*	Реализация параллельного алгоритма, *мс*
1	37207	27549	3516
2	36992	27491	3543
3	37098	27543	3547
4	37008	27472	3545
5	37038	27458	3490
6	36975	27585	3547
7	37050	27502	3502
8	36949	27514	3530
9	37036	27528	3495
10	37102	27444	3535
Среднее время	37045.5	27508.6	3525.0
Ускорение	1	1.34	10.51

Таблица 2 – Полученные времена исполнения и относительные ускорения различных версий алгоритмов.

Выводы

В данной работе получены следующие результаты:

- На базе последовательного алгоритма создан параллельный алгоритм моделирования процессов, происходящих во временно-проекционной камере многоцелевого детектора коллайдера NICA.
- Изменён класс *TpcDigitizerTask* программного комплекса *MpdRoot*, моделирующий временно-проекционную камеру многоцелевого детектора MPD коллайдера NICA, для возможности использования параллельного алгоритма с применением технологии NVIDIA CUDA.
- Изменено определение функции в файле *TMath.h* в системе Root для возможности компиляции программного комплекса *MpdRoot* компилятором языка C++ NVCC поддерживающим технологию NVIDIA CUDA.
- Изменен файл управления сборкой системы MpdRoot для возможности компиляции класса *TpcDigitizerTask* с использованием технологии NVIDIA CUDA.
- Произведено сравнение ускорения работы последовательной и параллельной реализации класса *TpcDigitizerTask*.

Работа была представлена на XVII научной конференции молодых учёных и специалистов (ОМУС-2013) [17] и была отправлена на публикацию в сборнике трудов. По итогам конференции также было получено направление на публикацию в письма в журнал «Физика элементарных частиц и атомного ядра» .

Используемая литература

Список литературы

1. *Gyulassy M.* The QGP Discovered at RHIC [Электронный ресурс].— Режим доступа: http://arxiv.org/pdf/nucl-th/0403032.

2. *Wilson T.* Super Proton Synchrotron marks its 25th birthday [Электронный ресурс].— Режим доступа: http://cerncourier.com/cws/article/cern/28470.

3. *Collaboration, RHIC.* Hunting the Quark Gluon Plasma (Formal Report) / RHIC Collaboration, New York:BNL, 2005 - 361 pp.

4. *MPD, Коллаборация.* Многоцелевой детектор MPD для изучения столкновений тяжелых ионов на ускорителе NICA (Концептуальный дизайн-проект) / Коллаборация MPD. — Дубна: ОИЯИ, 2010. — 224 с.

5. Root [Электронный ресурс]. — Режим доступа: http://root.cern.ch.

6. FairRoot [Электронный ресурс]. — Режим доступа: http://fairroot.gsi.de.

7. MpdRoot [Электронный ресурс]. — Режим доступа: http://mpd.jinr.ru.

8. CERN [Электронный ресурс]. — Режим доступа: http://home.web.cern.ch.

9. FAIR – Facility for Antiproton and Ion Research in Europe GmbH [Электронный ресурс]. — Режим доступа: http://www.fair-center.eu.

10. *Sissakian, A. N.* Design and Construction of Nuclotron-based Ion Collider fAcility (NICA), Conceptual design report / A. N. Sissakian [et al.]. — Dubna: JINR, 2008. — 149 pp.

11. NVIDIA. CUDA C Programming Guide [Электронный ресурс].— http://docs.nvidia.com/cuda-c-programming-guide/index.html.

12. История видеокарт [Электронный ресурс].— http://antonkozlov.ru/istoriya/istoriya-videokart-1.html.

13. Geant4 [Электронный ресурс]. — Режим доступа: http://geant4.cern.ch.

14. GNU. The GNU C++ Library [Электронный ресурс]. — http://gcc.gnu.org/onlinedocs/libstdc++.

15. NVIDIA. CURAND Library [Электронный ресурс].— http://docs.nvidia.com/cuda/curand/index.html.

16. UrQMD [Электронный ресурс]. — Режим доступа: http://urqmd.org.

17. *Соснов Д.Е.* Реализация алгоритмов реконструкции откликов частиц в TPC детектора NICA/MPD на гибридных вычислительных системах. / XVII научн. конф. молодых ученых и специалистов (ОМУС-2013) к 100-летию В.П.Джелепова. Дубна, 08-12 апр. 2013. Сб. аннот. докл.- Дубна, 2013.- С.17.

A Исходный код файла TpcDigitizerTask.h

TpcDigitizerTask.h

```
//-----------------------------------------------------------
//
// Description:
//      Tpc Digitizer reads array of MC points and produces TpcDigits
//
//
// Author List:
//      Dmitry Sosnov
//
//-----------------------------------------------------------

#ifndef TPCDIGITIZERTASK_HH
#define TPCDIGITIZERTASK_HH

// Base Class Headers ----------------

#include "FairTask.h"
#include <TNtuple.h>
#include "TpcPoint.h"
#include "TpcDigitizerQAHistograms.h"
#include "TLorentzVector.h"

class TClonesArray;
class TpcGas;

using namespace std;

#ifdef __CUDACC__ //if compiled with nvcc compiler
  #include <cuda.h> //required cuda header
  #include <curand_kernel.h> //required curand header
  #define BLOCK_SIZE 256 //Constant block size; TODO make it variabled
  #define CUDA_CAPABILITY_REQUIRE 20 //Cuda compute capability.
#else
/*Some defines for compiling with no-nvcc compiler*/
  #define __device__
```

```
36    #define __host__
37    #define __global__
38    #define __shared__
39    #define __constant__
40  #endif
41
42  struct CudaTpcLorenzVector {
43    Double_t X, Y, Z, T;
44  };
45
46  class ForStore: public TNamed {
47  public:
48
49    ForStore();
50    virtual ~ForStore();
51
52    Float_t ****fBuffer;
53  };
54
55  class TpcDigitizerTask: public FairTask {
56  public:
57
58    // Constructors/Destructors ---------
59    TpcDigitizerTask();
60    virtual ~TpcDigitizerTask();
61
62    Bool_t isSubtrackInInwards(const TpcPoint *p1, const TpcPoint *p2);
63
64    void SetPrimBranchName(const TString& name) {
65      fInputBranchName = name;
66    }
67
68    void SetPersistence(Bool_t opt = kTRUE) {
69      fPersistence = opt;
70    }
71
72    void SetAttach(Bool_t opt = kTRUE) {
73      fAttach = opt;
74    }
```

```
75
76    void SetDiffuse(Bool_t opt = kTRUE) {
77      fDiffuse = opt;
78    }
79
80    void SetDistort(Bool_t opt = kTRUE) {
81      fDistort = opt;
82    }
83
84    void SetDebug(Bool_t opt = kTRUE) {
85      fPrintDebugInfo = opt;
86    }
87
88    void SetMakeQA(Bool_t opt = kFALSE) {
89      fMakeQA = opt;
90    }
91
92    void SetPackSize(UInt_t packSize = 1E5) {
93      fPackSize = packSize;
94    }
95
96    void SetUseCuda(Bool_t opt = kTRUE) {
97      fUseCuda = opt;
98    }
99
100   virtual InitStatus Init();
101   virtual void Exec(Option_t* opt);
102   virtual void Finish();
103
104 private:
105   /*Distribute problems to GPU devices*/
106   __host__ void DistributorCuda(vector<CudaTpcLorenzVector> &electronVect,
        UInt_t threadDivide = 1E5);
107   /*Find lighted by avalanche pads (GPU version)*/
108   static __device__ void GetAreaCuda(Float_t xEll, Float_t yEll, UInt_t *padIDs,
        UInt_t *rowIDs, Float_t *amps, Float_t *ampSumOut,
109       Int_t *cudaFNumOfPadsInRow, UInt_t totalThreadCount, UInt_t index);
110   /*find amplitude by avalanche on pad (GPU version)*/
```

```
111     static __device__ Float_t CalculatePadResponseCuda(UInt_t padID, UInt_t rowID,
          Float_t x, Float_t y, Int_t *cudaFNumOfPadsInRow);
112     /*Distribute problems to CPU devices*/
113     __host__ void DistributorCPU(vector<CudaTpcLorenzVector> &electronVect);
114     /*Find lighted by avalanche pads (CPU version)*/
115     __host__ void GetAreaCPU(Float_t xEll, Float_t yEll, vector<UInt_t> &padIDs,
          vector<UInt_t> &rowIDs, vector<Float_t> &amps, Float_t &ampSumOut);
116     /*find amplitude by avalanche on pad (CPU version)*/
117     __host__ Float_t CalculatePadResponseCPU(UInt_t padID, UInt_t rowID, Float_t x
          , Float_t y);
118
119   #ifdef __CUDACC__ //if not NVCC compiler we haven't any cuda kernels
120     /*Kernel that make drift and call other __device__ functions*/
121     friend __global__ void driftKernelCuda(CudaTpcLorenzVector*
          electronPositionArray, curandState *stateGlobal, UInt_t totalThreadCount,
122           UInt_t maxLightedPadsCount, UInt_t *cudaLightedPadsOut, UInt_t *
              cudaLightedRowsOut, Float_t *cudaAmpsOut, UInt_t *cudaCurSectIDOut,
123           UInt_t *cudaCurTimeIDOut, Float_t *cudaAmpSumOut, Int_t *
              cudaFNumOfPadsInRow);
124   #endif
125
126    private:
127
128     // Private Data Members ------------
129     TString fInputBranchName;
130     TString fOutputBranchName;
131     TClonesArray* fMCPointArray;
132     ForStore *store;
133
134     TpcGas* fGas; // pointer to gas system
135     Float_t fGain; // coefficient for avalanches
136     Float_t zCathode; // length of TPC. TODO: get from geometry
137
138     Float_t ****fDigitsArray; //output array of digital signals
139
140     Int_t *fNumOfPadsInRow;
141
142     UInt_t nSectors;
143     UInt_t nTimeBackets;
```

```
144     UInt_t nRows;
145     UInt_t nInRows;
146     UInt_t nOutRows;
147 
148     Float_t r_min;
149 
150     Float_t fSpread; // sigma for pad response function.
151     Float_t fRadius; // cutoff radius for pad response function
152 
153     Float_t pwIn; //inner pad width
154     Float_t phIn; //inner pad hight
155 
156     Float_t pwOut; //outer pad width
157     Float_t phOut; //outer pad hight
158 
159     TpcDigitizerQAHistograms *fHisto; //class for QA generation
160 
161     Bool_t fIsHistogramsInitialized;
162     Bool_t fMakeQA;
163     Bool_t fPersistence;
164     Bool_t fAttach;
165     Bool_t fDiffuse;
166     Bool_t fDistort;
167     Bool_t fPrintDebugInfo;
168     Bool_t fUseCuda;
169 
170     UInt_t fPackSize; //Size for distribute in CUDA version
171 
172     // Private Methods ——————————————
173  public:
174  ClassDef(TpcDigitizerTask, 5)
175 
176  };
177 
178 #endif
```

B Исходный код файла TpcDigitizerTask.cu

TpcDigitizerTask.cu

```
1  //------------------------------------------------------------------
2  //
3  // Description:
4  //      Implementation of class TpcDigitizerTask
5  //      see TpcDigitizerTask.h for details
6  //
7  // Author List:
8  //      Dmitry Sosnov
9  //
10 //------------------------------------------------------------------
11
12 // Panda Headers ----------------------
13
14 // This Class' Header ------------------
15 #include "TpcDigitizerTask.h"
16
17 // C/C++ Headers ----------------------
18 #include <math.h>
19 #include <iostream>
20 #include <vector>
21 #include <algorithm>
22
23 #include "FairRunAna.h"
24 #include "FairEventHeader.h"
25 #include "TpcPoint.h"
26 #include "TLorentzVector.h"
27
28 // Collaborating Class Headers --------
29 #include "FairRootManager.h"
30 #include <TGeoManager.h>
31 #include <TGeoTube.h>
32 #include <TNtuple.h>
33 #include "TClonesArray.h"
34 #include "TpcGas.h"
35 #include "TRandom.h"
```

```
36 #include "TMath.h"
37 #include "TpcPrimaryCluster.h"
38 #include "TpcSector.h"
39 #include "TpcDriftedElectron.h"
40 #include "TSystem.h"
41 #include "TaskHelpers.h"
42 #include "TpcAvalanche.h"
43
44 using namespace std;
45 using namespace TMath;
46
47 #ifdef __CUDACC__
48 __constant__ Float_t cudaZCathode;
49 __constant__ Bool_t cudaFAttach, cudaFDiffuse, cudaFDistort;
50 __constant__ UInt_t cudaNSectors;
51 __constant__ UInt_t cudaNTimeBackets;
52 __constant__ Float_t cudaR_min;
53 __constant__ Double_t cudaFGasK, cudaFGasDt, cudaFGasDl, cudaFGasVDrift;
54 __constant__ Float_t cudaFSpread;
55 __constant__ Float_t cudaFRadius;
56 __constant__ Float_t cudaFGain;
57 __constant__ Float_t cudaPwIn; //inner pad width
58 __constant__ Float_t cudaPhIn; //inner pad hight
59 __constant__ Float_t cudaPwOut; //outer pad width
60 __constant__ Float_t cudaPhOut; //outer pad hight
61 __constant__ UInt_t cudaNRows;
62 __constant__ UInt_t cudaNInRows;
63 __constant__ UInt_t cudaNOutRows;
64 __global__ void setupCudaRandom(curandState *state, time_t inittime);
65 __global__ void driftKernelCuda(CudaTpcLorenzVector* electronPositionArray,
      curandState *stateGlobal, UInt_t totalThreadCount,
66     UInt_t maxLightedPadsCount, UInt_t *cudaLightedPadsOut, UInt_t *
         cudaLightedRowsOut, Float_t *cudaAmpsOut, UInt_t *cudaCurSectIDOut,
67     UInt_t *cudaCurTimeIDOut, Float_t *cudaAmpSumOut, Int_t *cudaFNumOfPadsInRow
         );
68 #endif
69
70
71 ForStore::ForStore() {
```

```
72    TpcSector* sector = new TpcSector();
73
74    UInt_t nSect = sector->GetNSectors();
75    UInt_t nTimeBins = sector->GetNTimeBins();
76    UInt_t nRows = sector->GetNumRows();
77    Int_t *fNumPadsInRow = sector->GetArrayPadsInRow();
78
79    fBuffer = new Float_t***[nSect];
80    for (UInt_t iSec = 0; iSec < nSect; ++iSec) {
81      fBuffer[iSec] = new Float_t**[nRows];
82      for (UInt_t iRows = 0; iRows < nRows; ++iRows) {
83        fBuffer[iSec][iRows] = new Float_t*[fNumPadsInRow[iRows] * 2];
84        for (UInt_t iPads = 0; iPads < fNumPadsInRow[iRows] * 2; ++iPads) {
85          fBuffer[iSec][iRows][iPads] = new Float_t[nTimeBins];
86          for (UInt_t iTime = 0; iTime < nTimeBins; ++iTime) {
87            fBuffer[iSec][iRows][iPads][iTime] = 0.0;
88          }
89        }
90      }
91    }
92  }
93
94  ForStore::~ForStore() {
95    TpcSector* sector = new TpcSector();
96    UInt_t nSect = sector->GetNSectors();
97    UInt_t nRows = sector->GetNumRows();
98    Int_t *fNumPadsInRow = sector->GetArrayPadsInRow();
99
100   for (UInt_t iSec = 0; iSec < nSect; ++iSec) {
101     for (UInt_t iRows = 0; iRows < nRows; ++iRows) {
102       for (UInt_t iPads = 0; iPads < fNumPadsInRow[iRows] * 2; ++iPads) {
103         delete[] fBuffer[iSec][iRows][iPads];
104       }
105       delete[] fBuffer[iSec][iRows];
106     }
107     delete[] fBuffer[iSec];
108   }
109   delete[] fBuffer;
110 }
```

```
111
112 #ifdef __CUDACC__
113 __global__ void setupCudaRandom(curandState *state, time_t inittime) { //
      Initialaise cuda random
114   UInt_t index = blockIdx.x * blockDim.x + threadIdx.x;
115   curand_init(inittime, index, 0, &state[index]);
116 }
117 #endif
118
119 /*Constructor*/
120 TpcDigitizerTask::TpcDigitizerTask() :
121     fPersistence(kTRUE), fAttach(kFALSE), fDiffuse(kFALSE), fDistort(kFALSE),
        fPrintDebugInfo(kFALSE), fIsHistogramsInitialized(kFALSE), fMakeQA(
122         kFALSE), fHisto(0), fPackSize(1E5), fUseCuda(kTRUE) {
123   fInputBranchName = "TpcPoint";
124   fOutputBranchName = "TpcDigits";
125
126   string tpcGasFile = gSystem->Getenv("VMCWORKDIR");
127   tpcGasFile += "/geometry/Ar-90_CH4-10.asc";
128   fGas = new TpcGas(tpcGasFile, 130);
129 }
130
131 /*Destructor*/
132 TpcDigitizerTask::~TpcDigitizerTask() {
133   delete fGas;
134 }
135
136 InitStatus TpcDigitizerTask::Init() {
137
138   //Get ROOT Manager
139   FairRootManager* ioman = FairRootManager::Instance();
140
141   if (!ioman) {
142     cout << "\n-E- [TpcDigitizerTask::Init]: RootManager not instantiated!" <<
        endl;
143     return kFATAL;
144   }
145   fMCPointArray = (TClonesArray*) ioman->GetObject(fInputBranchName);
146
```

```
147     TpcSector* sector = new TpcSector();
148     nTimeBackets = sector->GetNTimeBins();
149     nSectors = sector->GetNSectors();
150     pwIn = sector->GetInnerPadWidth();
151     pwOut = sector->GetOuterPadWidth();
152     phIn = sector->GetInnerPadHeight();
153     phOut = sector->GetOuterPadHeight();
154     nRows = sector->GetNumRows();
155     nInRows = sector->GetNumInnerRows();
156     nOutRows = sector->GetNumOuterRows();
157     r_min = 35.0; //27.0; // FIXME!!!
158
159     fNumOfPadsInRow = sector->GetArrayPadsInRow();
160     if (fPrintDebugInfo) {
161       cout << "Number of pads in every rows is ";
162       for (UInt_t k = 0; k < nRows; ++k)
163         cout << fNumOfPadsInRow[k] * 2 << " ";
164       cout << endl;
165     }
166
167     //memory allocating for output array
168     fDigitsArray = new Float_t***[nSectors];
169     for (UInt_t iSec = 0; iSec < nSectors; ++iSec) {
170       fDigitsArray[iSec] = new Float_t**[nRows];
171       for (UInt_t iRow = 0; iRow < nRows; ++iRow) {
172         fDigitsArray[iSec][iRow] = new Float_t*[fNumOfPadsInRow[iRow] * 2];
173         for (UInt_t iPad = 0; iPad < fNumOfPadsInRow[iRow] * 2; ++iPad) {
174           fDigitsArray[iSec][iRow][iPad] = new Float_t[nTimeBackets];
175           for (UInt_t iTime = 0; iTime < nTimeBackets; ++iTime) {
176             fDigitsArray[iSec][iRow][iPad][iTime] = 0.0;
177           }
178         }
179       }
180     }
181
182     store = new ForStore();
183     ioman->Register(fOutputBranchName, "TPC", store, fPersistence);
184
185     zCathode = sector->GetLength(); //cm
```

```
186     fGain = 5000; //electrons
187     fSpread = 0.196; // cm
188     fRadius = fSpread * 3;
189
190     if (!fIsHistogramsInitialized && fMakeQA) {
191       fHisto = new TpcDigitizerQAHistograms();
192       fHisto->Initialize();
193       fIsHistogramsInitialized = true;
194     }
195
196     cout << "-I- TpcDigitizerTask: Intialisation successfull." << endl << endl;
197     return kSUCCESS;
198   }
199
200   void TpcDigitizerTask::Exec(Option_t* opt) {
201
202     cout << "TpcDigitizer::Exec started" << endl;
203
204     for (UInt_t iSec = 0; iSec < nSectors; ++iSec) {
205       for (UInt_t iRow = 0; iRow < nRows; ++iRow) {
206         for (UInt_t iPad = 0; iPad < fNumOfPadsInRow[iRow] * 2; ++iPad) {
207           for (UInt_t iTime = 0; iTime < nTimeBackets; ++iTime) {
208             fDigitsArray[iSec][iRow][iPad][iTime] = 0.0;
209           }
210         }
211       }
212     }
213
214     Int_t nPoints = fMCPointArray->GetEntriesFast();
215     if (nPoints < 2) {
216       Warning("TpcDigitizerTask::Exec", "Not enough Hits in TPC for Digitization
            (<2)");
217       return;
218     }
219
220
221     TpcPoint* curPoint;
222     TpcPoint* prePoint = (TpcPoint*) fMCPointArray->At(0);
223     Float_t dE = 0.0; //energy loss
```

```
224     UInt_t qTotal = 0; //sum of clusters charges (=sum of electrons between two
            TpcPoints)
225     UInt_t qCluster = 0; //charge of cluster (= number of electrons)
226     TLorentzVector curPointPos; // coordinates for current TpcPoint
227     TLorentzVector prePointPos; // coordinates for previous TpcPoint
228     TLorentzVector diffPointPos; // steps for clusters creation
229     //TVector3 diffuse; // vector of diffuse for every coordinates
230     //TLorentzVector electronPos; // coordinates for created electrons
231     TLorentzVector clustPos; // coordinates for created clusters
232     vector<UInt_t> clustArr; // vector of clusters between two TpcPoints
233     vector<CudaTpcLorenzVector> electronVect; //vector of start electron positions
234
235     if (fPrintDebugInfo) {
236       cout << "Number of MC points is " << nPoints << endl << endl;
237     }
238     for (UInt_t i = 1; i < nPoints; i++) {
239       curPoint = (TpcPoint*) fMCPointArray->At(i);
240       //check if hits are on the same track
241       if (curPoint->GetTrackID() == prePoint->GetTrackID() && !isSubtrackInInwards
            (prePoint, curPoint)) {
242
243         dE = curPoint->GetEnergyLoss() * 1E9; //convert from GeV to eV
244         if (dE < 0) {
245           Error("TpcDigitizerTask::Exec", "Negative Energy loss!");
246           continue;
247         }
248
249         curPointPos.SetXYZT(curPoint->GetX(), curPoint->GetY(), curPoint->GetZ(),
              curPoint->GetTime());
250         prePointPos.SetXYZT(prePoint->GetX(), prePoint->GetY(), prePoint->GetZ(),
              prePoint->GetTime());
251         if ((curPointPos.T() < 0) || (prePointPos.T() < 0)) {
252           Error("TpcDigitizerTask::Exec", "Negative Time!");
253           continue;
254         }
255
256         diffPointPos = curPointPos - prePointPos; //differences between two points
              by coordinates
257         diffPointPos *= (1 / diffPointPos.Vect().Mag()); //directional cosines
```

```
258
259         qTotal = (UInt_t) floor(fabs(dE / fGas->W()));
260
261         //while still charge not used-up distribute charge into next cluster
262         while (qTotal > 0) {
263           //roll dice for next cluster
264           qCluster = fGas->GetRandomCSUniform();
265           if (qCluster > qTotal) qCluster = qTotal;
266           qTotal -= qCluster;
267           clustArr.push_back(qCluster);
268         } // finish loop for cluster creation
269
270         diffPointPos *= (diffPointPos.Vect().Mag() / clustArr.size()); //now here
              are steps between clusters by coordinates
271         clustPos = curPointPos;
272         for (UInt_t iClust = 0; iClust < clustArr.size(); ++iClust) {
273           clustPos += diffPointPos;
274           for (UInt_t iEll = 0; iEll < clustArr.at(iClust); ++iEll) {
275             electronVect.push_back((CudaTpcLorenzVector) {clustPos.X(),clustPos.Y
                  (),clustPos.Z(),clustPos.T()});
276           }
277         }
278       } //end check for same track
279       prePoint = curPoint;
280       clustArr.clear();
281       clustArr.resize(0);
282     } // finish loop over GHits
283
284 #ifdef __CUDACC__
285     if (fUseCuda) {
286       DistributorCuda(electronVect, fPackSize);
287     } else {
288       DistributorCPU(electronVect);
289     }
290 #else
291     DistributorCPU(electronVect);
292 #endif
293     electronVect.clear(); //just in case
294     for (UInt_t iSec = 0; iSec < nSectors; ++iSec) {
```

```
295      for (UInt_t iRows = 0; iRows < nRows; ++iRows) {
296        for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[iRows] * 2; ++iPads) {
297          for (UInt_t iTime = 0; iTime < nTimeBackets; ++iTime) {
298            store->fBuffer[iSec][iRows][iPads][iTime] = fDigitsArray[iSec][iRows][
                   iPads][iTime];
299          }
300        }
301      }
302    }
303    cout << "TpcDigitizer::Exec finished" << endl << endl;
304  }
305
306  /*
307   * GPU (CUDA)
308   */
309  #ifdef __CUDACC__
310  __host__ void TpcDigitizerTask::DistributorCuda(vector<CudaTpcLorenzVector> &
         electronVect, UInt_t threadDivideBase) {
311    UInt_t electronGlobalCount = electronVect.size();
312
313    if (electronGlobalCount == 0) {
314      Error("TpcDigitizerTask::DistributorCuda", "No electrons at all!!!");
315      return;
316    }
317    if (fPrintDebugInfo) {
318      printf("Number of electrons is %u\n", electronGlobalCount);
319    }
320    if (threadDivideBase == 0) {
321      threadDivideBase = 1;
322    }
323    if (electronGlobalCount <= threadDivideBase) {
324      threadDivideBase = electronGlobalCount;
325    }
326    int deviceCount;
327    vector<UInt_t> devices; //vector of suitable devices
328    vector<UInt_t> warpSizes; //vector of warp sizes of suitable devices
329    cudaDeviceProp cudaProp; //device properties
330    cudaSetDeviceFlags(cudaDeviceMapHost); //for device pointer to any mapped
           memory (just in case?)
```

```
331     cudaGetDeviceCount(&deviceCount);
332     for (UInt_t dev = 0; dev < deviceCount; ++dev) { //fill vectors
333       cudaGetDeviceProperties(&cudaProp, dev);
334       if (cudaProp.major * 10 + cudaProp.minor >= CUDA_CAPABILITY_REQUIRE) {
335         devices.push_back(dev);
336         warpSizes.push_back(cudaProp.warpSize);
337       }
338     }
339     if (devices.size() != 0) {//If it is some devices
340       deviceCount = devices.size();
341       if (fPrintDebugInfo) {
342         printf("Found %i suitable CUDA devices\n", deviceCount);
343       }
344       cudaStream_t *stream = (cudaStream_t*) malloc(deviceCount * sizeof(
            cudaStream_t));
345       for (int dev = 0; dev < deviceCount; ++dev) { //Create streams for
            calculating on multiply gpu - stream for gpu
346         cudaSetDevice(devices[dev]);
347         cudaStreamCreate(&stream[dev]);
348       }
349       UInt_t *threadDivide = (UInt_t*) malloc(deviceCount * sizeof(UInt_t)); //
            size of pack of electrons for kernel
350       UInt_t *cudaThreadPerBlock = (UInt_t*) malloc(deviceCount * sizeof(UInt_t));
            //Thread per block (second parameter in kernel start parameters)
351       UInt_t sum_threadDivide = 0; //sum of threadDivide's
352       for (int dev = 0; dev < deviceCount; ++dev) {
353         threadDivide[dev] = ceil((Double_t) threadDivideBase / warpSizes[dev]) *
              warpSizes[dev];
354         cudaThreadPerBlock[dev] = BLOCK_SIZE; //cudaProp.maxThreadsPerBlock; //
              TODO variable block size
355         if (cudaThreadPerBlock[dev] > threadDivide[dev]) cudaThreadPerBlock[dev] =
              threadDivide[dev];
356         sum_threadDivide += threadDivide[dev];
357       }
358
359       UInt_t maxLightedPadsCount = 4.0 * max(Ceil(pwIn / fRadius) * Ceil(phIn /
          fRadius), Ceil(pwOut / fRadius) * Ceil(phOut / fRadius)); //maximum count
           of pads, may be lighted by avalanche
```

```
360     Double_t fGasK = fGas->k(), fGasDt = fGas->Dt(), fGasDl = fGas->Dl(),
            fGasVDrift = fGas->VDrift(); //Some variables for constants
361     cudaError_t *err = (cudaError_t*) malloc(deviceCount * sizeof(cudaError_t));
             //Array of errors
362     time_t inittime;
363     CudaTpcLorenzVector **cudaElectronPositionArray = (CudaTpcLorenzVector**)
            malloc(deviceCount * sizeof(CudaTpcLorenzVector*));
364     /* arrays(by devices) of arrays/variables */
365     Int_t **cudaFNumOfPadsInRow = (Int_t**) malloc(deviceCount * sizeof(Int_t*))
            ; //data from fNumOfPadsInRow on devices
366     UInt_t **cudaLightedPadsOut = (UInt_t**) malloc(deviceCount * sizeof(UInt_t
            *)); //Array of lighted by electron pads (kernel output)
367     UInt_t **cudaLightedRowsOut = (UInt_t**) malloc(deviceCount * sizeof(UInt_t
            *)); //Array of rows of lighted by electron pads (kernel output)
368     Float_t **cudaAmpsOut = (Float_t**) malloc(deviceCount * sizeof(Float_t*));
            //Array of amplitudes on pads (kernel output)
369     Float_t **cudaAmpNormOut = (Float_t**) malloc(deviceCount * sizeof(Float_t*)
            ); //Normalization coefficient for amplitudes (\sum(amplitudes)*fGain)
370     UInt_t **cudaCurSectIDOut = (UInt_t**) malloc(deviceCount * sizeof(UInt_t*))
            ; //Sector of avalanche
371     UInt_t **cudaCurTimeIDOut = (UInt_t**) malloc(deviceCount * sizeof(UInt_t*))
            ; //Time of avalanche
372     UInt_t *cudaLightedPads, *cudaLightedRows, *cudaCurSectID, *cudaCurTimeID;
            //Arrays for gather kernel outs
373     Float_t *cudaAmps, *cudaAmpNorm; //Arrays for gather kernel outs
374     curandState **devStates = (curandState**) malloc(deviceCount * sizeof(
            curandState*)); // curand states arrays
375
376     for (int dev = 0; dev < deviceCount; ++dev) {//Mallocs
377       cudaSetDevice(devices[dev]);
378       cudaMalloc(&devStates[dev], threadDivide[dev] * sizeof(curandState));
379       cudaMalloc(&cudaFNumOfPadsInRow[dev], sizeof(Int_t) * nRows);
380       cudaMalloc(&cudaElectronPositionArray[dev], sizeof(CudaTpcLorenzVector) *
              threadDivide[dev]);
381       cudaMalloc(&cudaLightedPadsOut[dev], maxLightedPadsCount * threadDivide[
              dev] * sizeof(UInt_t));
382       cudaMalloc(&cudaLightedRowsOut[dev], maxLightedPadsCount * threadDivide[
              dev] * sizeof(UInt_t));
```

```
383      cudaMalloc(&cudaAmpsOut[dev], maxLightedPadsCount * threadDivide[dev] *
           sizeof(Float_t));
384      cudaMalloc(&cudaAmpNormOut[dev], threadDivide[dev] * sizeof(Float_t));
385      cudaMalloc(&cudaCurSectIDOut[dev], threadDivide[dev] * sizeof(UInt_t));
386      cudaMalloc(&cudaCurTimeIDOut[dev], threadDivide[dev] * sizeof(UInt_t));
387    }
388    for (int dev = 0; dev < deviceCount; ++dev) {//Memcpys
389      cudaSetDevice(devices[dev]);
390      cudaMemcpyToSymbolAsync(cudaZCathode, &zCathode, sizeof(zCathode), 0,
           cudaMemcpyHostToDevice, stream[dev]);
391      cudaMemcpyToSymbolAsync(cudaFAttach, &fAttach, sizeof(fAttach), 0,
           cudaMemcpyHostToDevice, stream[dev]);
392      cudaMemcpyToSymbolAsync(cudaFDiffuse, &fDiffuse, sizeof(fDiffuse), 0,
           cudaMemcpyHostToDevice, stream[dev]);
393      cudaMemcpyToSymbolAsync(cudaFDistort, &fDistort, sizeof(fDistort), 0,
           cudaMemcpyHostToDevice, stream[dev]);
394      cudaMemcpyToSymbolAsync(cudaNSectors, &nSectors, sizeof(nSectors), 0,
           cudaMemcpyHostToDevice, stream[dev]);
395      cudaMemcpyToSymbolAsync(cudaNTimeBackets, &nTimeBackets, sizeof(
           nTimeBackets), 0, cudaMemcpyHostToDevice, stream[dev]);
396      cudaMemcpyToSymbolAsync(cudaR_min, &r_min, sizeof(r_min), 0,
           cudaMemcpyHostToDevice, stream[dev]);
397      cudaMemcpyToSymbolAsync(cudaFGasK, &fGasK, sizeof(fGasK), 0,
           cudaMemcpyHostToDevice, stream[dev]);
398      cudaMemcpyToSymbolAsync(cudaFGasDt, &fGasDt, sizeof(fGasDt), 0,
           cudaMemcpyHostToDevice, stream[dev]);
399      cudaMemcpyToSymbolAsync(cudaFGasDl, &fGasDl, sizeof(fGasDl), 0,
           cudaMemcpyHostToDevice, stream[dev]);
400      cudaMemcpyToSymbolAsync(cudaFGasVDrift, &fGasVDrift, sizeof(fGasVDrift),
           0, cudaMemcpyHostToDevice, stream[dev]);
401      cudaMemcpyToSymbolAsync(cudaFSpread, &fSpread, sizeof(fSpread), 0,
           cudaMemcpyHostToDevice, stream[dev]);
402      cudaMemcpyToSymbolAsync(cudaFRadius, &fRadius, sizeof(fRadius), 0,
           cudaMemcpyHostToDevice, stream[dev]);
403      cudaMemcpyToSymbolAsync(cudaFGain, &fGain, sizeof(fGain), 0,
           cudaMemcpyHostToDevice, stream[dev]);
404      cudaMemcpyToSymbolAsync(cudaPwIn, &pwIn, sizeof(pwIn), 0,
           cudaMemcpyHostToDevice, stream[dev]);
```

```
405       cudaMemcpyToSymbolAsync(cudaPhIn, &phIn, sizeof(phIn), 0,
              cudaMemcpyHostToDevice, stream[dev]);
406       cudaMemcpyToSymbolAsync(cudaPwOut, &pwOut, sizeof(pwOut), 0,
              cudaMemcpyHostToDevice, stream[dev]);
407       cudaMemcpyToSymbolAsync(cudaPhOut, &phOut, sizeof(phOut), 0,
              cudaMemcpyHostToDevice, stream[dev]);
408       cudaMemcpyToSymbolAsync(cudaNRows, &nRows, sizeof(nRows), 0,
              cudaMemcpyHostToDevice, stream[dev]);
409       cudaMemcpyToSymbolAsync(cudaNInRows, &nInRows, sizeof(nInRows), 0,
              cudaMemcpyHostToDevice, stream[dev]);
410       cudaMemcpyToSymbolAsync(cudaNOutRows, &nOutRows, sizeof(nOutRows), 0,
              cudaMemcpyHostToDevice, stream[dev]);
411       cudaMemcpyAsync(cudaFNumOfPadsInRow[dev], fNumOfPadsInRow, sizeof(Int_t) *
               nRows, cudaMemcpyHostToDevice, stream[dev]);
412     }
413     cudaHostAlloc(&cudaLightedPads, maxLightedPadsCount * sum_threadDivide *
          sizeof(UInt_t), cudaHostAllocDefault);
414     cudaHostAlloc(&cudaLightedRows, maxLightedPadsCount * sum_threadDivide *
          sizeof(UInt_t), cudaHostAllocDefault);
415     cudaHostAlloc(&cudaAmps, maxLightedPadsCount * sum_threadDivide * sizeof(
          Float_t), cudaHostAllocDefault);
416     cudaHostAlloc(&cudaAmpNorm, sum_threadDivide * sizeof(Float_t),
          cudaHostAllocDefault);
417     cudaHostAlloc(&cudaCurSectID, sum_threadDivide * sizeof(UInt_t),
          cudaHostAllocDefault);
418     cudaHostAlloc(&cudaCurTimeID, sum_threadDivide * sizeof(UInt_t),
          cudaHostAllocDefault);
419
420     cudaDeviceSynchronize();
421     for (int dev = 0; dev < deviceCount; ++dev) {//Kernel for generating random
          states for random values
422       cudaSetDevice(devices[dev]);
423       inittime = time(0);
424       setupCudaRandom<<<( threadDivide[dev]+cudaThreadPerBlock[dev]-1)/
              cudaThreadPerBlock[dev],cudaThreadPerBlock[dev], 0, stream[dev]>>>(
              devStates[dev], inittime);
425       err[dev] = cudaGetLastError();
426     }
427     for (int i = 0; i < deviceCount; ++i) {
```

```
428       if (err[i] != cudaSuccess) {
429         printf("Initialize CUDA random sequences: Error %i\n", err[i]);
430         Error("TpcDigitizerTask::DistributorCuda", "CUDA random sequences
              initialize exited with errorcode %i.", err[i]);
431         exit(0);
432       } else {
433         if (fPrintDebugInfo) {
434           printf("Initialize CUDA random sequences: Success\n");
435         }
436       }
437     }
438
439     UInt_t *electronCount = (UInt_t*) malloc(deviceCount * sizeof(UInt_t)); //
          Count of real electrons for kernel (in iteration)
440     UInt_t *totalThreadCount = (UInt_t*) malloc(deviceCount * sizeof(UInt_t));
          //count of threads using for calculating electrons - it's better when
          count using threads multiply warpSize
441     CudaTpcLorenzVector* electronPositionArray; //Array of electrons using in
          iteration
442     UInt_t calculated = 0; //count of already calculated electrons
443     Int_t overflowIndex = -1; //index of end input electron array - stream(
          kernel) what calculate last batch set it stream number
444     UInt_t kernelDataOffset, arrayIndex; //Offset for kernel for gathered from
          kernels arrays; Index in arrays (lifhtedPads, lightedRows, Amps)
445     while (calculated < electronGlobalCount) {
446
447       for (UInt_t dev = 0; dev < deviceCount; ++dev) {//selection electrons for
            calculating in iteration by kernel(stream)
448         if ((overflowIndex >= 0) && (overflowIndex < dev)) continue;
449         cudaSetDevice(devices[dev]);
450         electronCount[dev] = min(threadDivide[dev], electronGlobalCount -
              calculated); //Count of electron for calculation in this iteration
451         if (electronCount[dev] == threadDivide[dev]) {
452           totalThreadCount[dev] = electronCount[dev];
453         } else {
454           totalThreadCount[dev] = Ceil((Double_t) electronCount[dev] / warpSizes
                [dev]) * warpSizes[dev];
455           for (int i = 0; i < totalThreadCount[dev] - electronCount[dev]; ++i) {
                //alignment of size by electronVect to totalThreadCount if tail
```

```
        less then totalThreadCount
456         electronVect.push_back(electronVect[calculated + electronCount[dev]
              - 1]);
457       }
458     }
459     electronPositionArray = &electronVect[calculated];
460     cudaMemcpyAsync(cudaElectronPositionArray[dev], electronPositionArray,
          sizeof(CudaTpcLorenzVector) * totalThreadCount[dev],
461       cudaMemcpyHostToDevice, stream[dev]);
462     if (fPrintDebugInfo) {
463       printf("Electrons from %u to %u (%u threads) at stream %u (device %u)\
            n", calculated, calculated + electronCount[dev], totalThreadCount[
            dev], dev, devices[dev]);
464     }
465     calculated += electronCount[dev];
466     if (calculated >= electronGlobalCount) overflowIndex = dev; //set
          overflow index
467   }
468
469   for (int dev = 0; dev < deviceCount; ++dev) {//Start calculation kernels
470     if ((overflowIndex >= 0) && (overflowIndex < dev)) continue;
471     cudaSetDevice(devices[dev]);
472     driftKernelCuda<<<(totalThreadCount[dev]+cudaThreadPerBlock[dev]-1)/
          cudaThreadPerBlock[dev],cudaThreadPerBlock[dev], 0, stream[dev]>>>(
          cudaElectronPositionArray[dev], devStates[dev], totalThreadCount[dev
          ], maxLightedPadsCount, cudaLightedPadsOut[dev], cudaLightedRowsOut[
          dev], cudaAmpsOut[dev], cudaCurSectIDOut[dev], cudaCurTimeIDOut[dev],
          cudaAmpNormOut[dev], cudaFNumOfPadsInRow[dev]);
473     err[dev] = cudaGetLastError();
474   }
475   for (int dev = 0; dev < deviceCount; ++dev) {
476     if ((overflowIndex >= 0) && (overflowIndex < dev)) continue;
477     if (err[dev] != cudaSuccess) {
478       printf("CUDA stream %u (device %u) status: Error %i\n", dev, devices[
            dev], err[dev]);
479       Error("TpcDigitizerTask::DistributorCuda", "CUDA stream %u (device %u)
            exited with errorcode %i.", dev, devices[dev], err[dev]);
480       exit(0);
481     } else {
```

```
482             if (fPrintDebugInfo) {
483               //printf("CUDA stream %u (device %u) status: Success\n", dev,
                      devices[dev]); //Exclude success output
484             }
485           }
486         }
487
488         kernelDataOffset = 0;
489         for (int dev = 0; dev < deviceCount; ++dev) {//copying data from device
490           if ((overflowIndex >= 0) && (overflowIndex < dev)) continue;
491           cudaSetDevice(devices[dev]);
492           cudaMemcpyAsync(&cudaLightedPads[kernelDataOffset * maxLightedPadsCount
                ], cudaLightedPadsOut[dev],
493                 maxLightedPadsCount * threadDivide[dev] * sizeof(UInt_t),
                      cudaMemcpyDeviceToHost, stream[dev]);
494           cudaMemcpyAsync(&cudaLightedRows[kernelDataOffset * maxLightedPadsCount
                ], cudaLightedRowsOut[dev],
495                 maxLightedPadsCount * threadDivide[dev] * sizeof(UInt_t),
                      cudaMemcpyDeviceToHost, stream[dev]);
496           cudaMemcpyAsync(&cudaAmps[kernelDataOffset * maxLightedPadsCount],
                cudaAmpsOut[dev], maxLightedPadsCount * threadDivide[dev] * sizeof(
                Float_t),
497                 cudaMemcpyDeviceToHost, stream[dev]);
498           cudaMemcpyAsync(&cudaAmpNorm[kernelDataOffset], cudaAmpNormOut[dev],
                threadDivide[dev] * sizeof(Float_t), cudaMemcpyDeviceToHost, stream[
                dev]);
499           cudaMemcpyAsync(&cudaCurSectID[kernelDataOffset], cudaCurSectIDOut[dev],
                 threadDivide[dev] * sizeof(UInt_t), cudaMemcpyDeviceToHost,
500                 stream[dev]);
501           cudaMemcpyAsync(&cudaCurTimeID[kernelDataOffset], cudaCurTimeIDOut[dev],
                 threadDivide[dev] * sizeof(UInt_t), cudaMemcpyDeviceToHost,
502                 stream[dev]);
503           kernelDataOffset += threadDivide[dev];
504         }
505         cudaDeviceSynchronize();
506
507         kernelDataOffset = 0;
508         arrayIndex = 0;
509         for (int dev = 0; dev < deviceCount; ++dev) {//fill out fDigitsArray array
```

```
510         if ((overflowIndex >= 0) && (overflowIndex < dev)) continue;
511         for (int j = 0; j < electronCount[dev]; ++j) {
512           for (int i = 0; i < maxLightedPadsCount; ++i) {
513             arrayIndex = kernelDataOffset * maxLightedPadsCount + i *
                    totalThreadCount[dev] + j;
514             if ((cudaAmps[arrayIndex] > 0) && (cudaAmpNorm[j + kernelDataOffset]
                    > 0)) {//amplitude and sum of amplitudes(divisor at cudaAmpNorm
                    computing) must be positive
515               fDigitsArray[cudaCurSectID[j + kernelDataOffset]][cudaLightedRows[
                      arrayIndex]][cudaLightedPads[arrayIndex]][cudaCurTimeID[j +
                      kernelDataOffset]] +=
516                   cudaAmps[arrayIndex] * cudaAmpNorm[j + kernelDataOffset];
517             }
518           }
519         }
520         kernelDataOffset += threadDivide[dev];
521       }
522     }
523     for (int i = 0; i < deviceCount; ++i) {//free composite pointers
524       cudaSetDevice(devices[i]);
525       cudaFree(cudaElectronPositionArray[i]);
526       cudaFree(cudaLightedPadsOut[i]);
527       cudaFree(cudaLightedRowsOut[i]);
528       cudaFree(cudaAmpsOut[i]);
529       cudaFree(cudaAmpNormOut[i]);
530       cudaFree(cudaCurSectIDOut[i]);
531       cudaFree(cudaCurTimeIDOut[i]);
532     }
533     cudaFreeHost(cudaLightedPadsOut);
534     cudaFreeHost(cudaLightedRowsOut);
535     cudaFreeHost(cudaAmpsOut);
536     cudaFreeHost(cudaAmpNormOut);
537     cudaFreeHost(cudaCurSectIDOut);
538     cudaFreeHost(cudaCurTimeIDOut);
539     cudaFreeHost(cudaLightedPads);
540     cudaFreeHost(cudaLightedRows);
541     cudaFreeHost(cudaAmps);
542     cudaFreeHost(cudaAmpNorm);
543     cudaFreeHost(cudaCurSectID);
```

```
544       cudaFreeHost(cudaCurTimeID);
545       for (int i = 0; i < deviceCount; ++i) {//delete streams
546         cudaSetDevice(devices[i]);
547         cudaStreamDestroy(stream[i]);
548       }
549       for (int dev = 0; dev < deviceCount; ++dev) {//reset devices - all from us
            removed
550         cudaSetDevice(devices[dev]);
551         cudaDeviceReset();
552       }
553     } else {//If it is no any device
554       Error("TpcDigitizerTask::DistributorCuda", "No suitable CUDA devices!");
555       if (fPrintDebugInfo) {
556         printf("Start calculating with CPU\n");
557       }
558       DistributorCPU(electronVect);
559     }
560   }
561   #endif
562
563   #ifdef __CUDACC__
564   /**
565    * driftKernelCuda - CUDA kernel that make drift and call other __device__
        functions
566    * parameters:
567    * electronPositionArray - array of electron positions (input)
568    * stateGlobal - CURAND states vector
569    * totalThreadCount - count of threads
570    * maxLightedPadsCount - maximum count of pads, may be lighted by avalanche
571    * cudaLightedPadsOut - array of pad lighted by avalanche
572    * cudaLightedRowsOut - array of rows of pads lighted by avalanche
573    * cudaAmpsOut - amplitudes on pads
574    * cudaCurSectIDOut - sectors of avalanches
575    * cudaCurTimeIDOut - time of avalanche
576    * cudaAmpNormOut - Normalization coefficient
577    * cudaFNumOfPadsInRow - pointer to fNumOfPadsInRow in global memory
578    */
579   __global__ void driftKernelCuda(CudaTpcLorenzVector* electronPositionArray,
        curandState *stateGlobal, UInt_t totalThreadCount,
```

```
580     UInt_t maxLightedPadsCount, UInt_t *cudaLightedPadsOut, UInt_t *
            cudaLightedRowsOut, Float_t *cudaAmpsOut, UInt_t *cudaCurSectIDOut,
581     UInt_t *cudaCurTimeIDOut, Float_t *cudaAmpNormOut, Int_t *
            cudaFNumOfPadsInRow) {
582   UInt_t index = blockIdx.x * blockDim.x + threadIdx.x;
583   if (index < totalThreadCount) {
584     __shared__ CudaTpcLorenzVector electronPos[BLOCK_SIZE]; //shared array with
            Vectors of electron position
585     electronPos[threadIdx.x] = electronPositionArray[index]; //copying electrons
586     __shared__ curandState stateShared[BLOCK_SIZE]; //shared curand states
587     stateShared[threadIdx.x] = stateGlobal[index]; //copying to shared from
            global
588     __shared__ CudaTpcLorenzVector diffuse[BLOCK_SIZE]; //Vector of diffusion
589     __shared__ UInt_t curSectID[BLOCK_SIZE]; //SectID in shared memory
590     __shared__ UInt_t curTimeID[BLOCK_SIZE]; //TimeID in shared memory
591     Float_t cudaPhiStep = (2 * CR_CUDART_PI / cudaNSectors) * 2;
592
593     Float_t driftl = cudaZCathode - fabs(electronPos[threadIdx.x].Z); // length
            for drifting
594     //If amplitude below zero - amplitude not add to out fDigitsArray
595     for (UInt_t i = 0; i < maxLightedPadsCount; ++i) {
596       cudaAmpsOut[index + totalThreadCount * i] = -1;
597     }
598     //attachment
599     if (cudaFAttach) {
600       if (exp(-driftl * cudaFGasK) < curand_uniform_double(&stateShared[
              threadIdx.x])) return;
601     }
602
603     //diffusion
604     diffuse[threadIdx.x] = (CudaTpcLorenzVector) {0, 0, 0, 0};
605     if (cudaFDiffuse) {
606       Float_t sqrtDrift = sqrt(driftl);
607       Float_t sigmat = cudaFGasDt * sqrtDrift;
608       Float_t sigmal = cudaFGasDl * sqrtDrift;
609       diffuse[threadIdx.x].X = curand_normal_double(&stateShared[threadIdx.x]) /
                sigmat;
610       diffuse[threadIdx.x].Y = curand_normal_double(&stateShared[threadIdx.x]) /
                sigmat;
```

```
611         diffuse[threadIdx.x].Z = curand_normal_double(&stateShared[threadIdx.x]) /
                sigmal;
612       }
613       //drift distortions
614       if (cudaFDistort) {
615         // TODO: to be implemented
616       }
617
618       electronPos[threadIdx.x].X += diffuse[threadIdx.x].X;
619       electronPos[threadIdx.x].Y += diffuse[threadIdx.x].Y;
620       electronPos[threadIdx.x].Z += diffuse[threadIdx.x].Z;
621       /*old
622        * //electronPos[threadIdx.x].T += (cudaZCathode - fabs(electronPos[
             threadIdx.x].Z)) / cudaFGasVDrift; // Do we need to use clustPos.T() ???
623        * //Float_t timeStep = (cudaZCathode / cudaNTimeBackets) / cudaFGasVDrift;
624        * //UInt_t curTimeID = (UInt_t) ((cudaZCathode / cudaFGasVDrift -
             electronPos[threadIdx.x].T) / timeStep); //To shared memory:
625        * //curTimeID[threadIdx.x] = (UInt_t) ((cudaZCathode / cudaFGasVDrift -
             electronPos[threadIdx.x].T) / ((cudaZCathode / cudaNTimeBackets) /
             cudaFGasVDrift));
626       */
627       curTimeID[threadIdx.x] = (UInt_t) ((fabs(electronPos[threadIdx.x].Z) -
            electronPos[threadIdx.x].T * cudaFGasVDrift) * cudaNTimeBackets  /
            cudaZCathode);
628       if (curTimeID[threadIdx.x] >= cudaNTimeBackets) return;
629       cudaCurTimeIDOut[index] = curTimeID[threadIdx.x];    //Will use global memory
             - shared very small for big maxLightedPadsCount values
630
631       Float_t globPhi = atan2(electronPos[threadIdx.x].Y, electronPos[threadIdx.x
            ].X); //angle in global coordinates
632       if (globPhi < 0) globPhi += 2 * CR_CUDART_PI; // TwoPi();
633       curSectID[threadIdx.x] = (UInt_t) (globPhi / cudaPhiStep + 0.5); //index of
            current sector
634       if (curSectID[threadIdx.x] == cudaNSectors / 2) curSectID[threadIdx.x] = 0;
635       Float_t sectPhi = curSectID[threadIdx.x] * cudaPhiStep; //const
636       if (electronPos[threadIdx.x].Z < 0.0) curSectID[threadIdx.x] += (
            cudaNSectors / 2);
637       cudaCurSectIDOut[index] = curSectID[threadIdx.x]; //Will use global memory -
             shared very small for big maxLightedPadsCount values
```

```
638
639       //local coordinates of electron (sector coordinates)
640       Float_t localX = -electronPos[threadIdx.x].X * sin(sectPhi) + electronPos[
              threadIdx.x].Y * cos(sectPhi);
641       Float_t localY = electronPos[threadIdx.x].X * cos(sectPhi) + electronPos[
              threadIdx.x].Y * sin(sectPhi) - cudaR_min;
642
643       //Will use global memory - shared very small for big maxLightedPadsCount
              values
644       TpcDigitizerTask::GetAreaCuda(localX, localY, cudaLightedPadsOut,
              cudaLightedRowsOut, cudaAmpsOut, cudaAmpNormOut, cudaFNumOfPadsInRow,
645           totalThreadCount, index); //Radius get from class variable
646     }
647   }
648   #endif
649
650   /**
651    *
652    */
653   __device__ void TpcDigitizerTask::GetAreaCuda(Float_t xEll, Float_t yEll, UInt_t
          *padIDs, UInt_t *rowIDs, Float_t *amps, Float_t *ampNormOut,
654       Int_t *cudaFNumOfPadsInRow, UInt_t totalThreadCount, UInt_t index) {
655     Float_t padW = 0.0, padH = 0.0;
656     Float_t y, x;
657     UInt_t pad = 0, row = 0;
658     Float_t amplithude;
659     UInt_t padCount = 0;
660     Float_t ampSum = 0.0;
661
662     if (fabs(yEll - cudaNInRows * cudaPhIn) < cudaFRadius) { //layer with both
          types of pads (inner and outer)
663       y = yEll - cudaFRadius;
664       x = xEll - cudaFRadius - cudaPwIn;
665       do {
666         x += cudaPwIn;
667         row = cudaNInRows - 1; //last inner row
668         if (x > 0.0) {
669           pad = Int_t(cudaFNumOfPadsInRow[row] + floor(x / cudaPwIn));
670         } else {
```

```
671           pad = Int_t(cudaFNumOfPadsInRow[row] - 1 + ceil(x / cudaPwIn));
672         }
673         if (pad >= cudaFNumOfPadsInRow[row] * 2) {
674           continue;
675         }
676         /*padIDs[padsCount] = pad;
677          rowIDs[padsCount] = row;
678          (padsCount)++;*/
679         amplithude = CalculatePadResponseCuda(pad, row, xEll, yEll,
               cudaFNumOfPadsInRow);
680         ampSum += amplithude;
681         amps[index + totalThreadCount * padCount] = amplithude;
682         padIDs[index + totalThreadCount * padCount] = pad;
683         rowIDs[index + totalThreadCount * padCount] = row;
684         (padCount)++;
685       } while (x < xEll + cudaFRadius);
686       y = yEll + cudaFRadius;
687       x = xEll - cudaFRadius - cudaPwOut;
688       do {
689         x += cudaPwOut;
690         row = cudaNInRows; //first outer row
691         if (x > 0.0) {
692           pad = Int_t(cudaFNumOfPadsInRow[row] + floor(x / cudaPwOut));
693         } else {
694           pad = Int_t(cudaFNumOfPadsInRow[row] - 1 + ceil(x / cudaPwOut));
695         }
696         if (pad >= cudaFNumOfPadsInRow[row] * 2) {
697           continue;
698         }
699         /*padIDs[padsCount] = pad;
700          rowIDs[padsCount] = row;
701          (padsCount)++;*/
702         amplithude = CalculatePadResponseCuda(pad, row, xEll, yEll,
               cudaFNumOfPadsInRow);
703         ampSum += amplithude;
704         amps[index + totalThreadCount * padCount] = amplithude;
705         padIDs[index + totalThreadCount * padCount] = pad;
706         rowIDs[index + totalThreadCount * padCount] = row;
707         (padCount)++;
```

```
708        } while (x < xEll + cudaFRadius);
709      } else {
710        if (yEll + cudaFRadius < cudaNInRows * cudaPhIn) { // inner pads
711          padW = cudaPwIn;
712          padH = cudaPhIn;
713        } else if (yEll - cudaFRadius > cudaNInRows * cudaPhIn) { //outer pads
714          padW = cudaPwOut;
715          padH = cudaPhOut;
716        }
717        x = xEll - cudaFRadius - padW;
718        do {
719          x += padW;
720          y = yEll - cudaFRadius - padH;
721          do {
722            y += padH;
723            row = (UInt_t) (y / padH);
724            if (x > 0.0) {
725              pad = Int_t(cudaFNumOfPadsInRow[row] + floor(x / padW));
726            } else {
727              pad = Int_t(cudaFNumOfPadsInRow[row] - 1 + ceil(x / padW));
728            }
729            if (row >= cudaNRows || pad >= cudaFNumOfPadsInRow[row] * 2) {
730              continue;
731            }
732            /*padIDs[padsCount] = pad;
733             rowIDs[padsCount] = row;
734             (padsCount)++;*/
735            amplithude = CalculatePadResponseCuda(pad, row, xEll, yEll,
                   cudaFNumOfPadsInRow);
736            ampSum += amplithude;
737            amps[index + totalThreadCount * padCount] = amplithude;
738            padIDs[index + totalThreadCount * padCount] = pad;
739            rowIDs[index + totalThreadCount * padCount] = row;
740            (padCount)++;
741          } while (y < yEll + cudaFRadius);
742        } while (x < xEll + cudaFRadius);
743      }
744      if (ampSum != 0) {
745        ampNormOut[index] = cudaFGain / ampSum;
```

```
746     } else {
747       ampNormOut[index] = -1;
748     }
749 }
750
751 __device__ Float_t TpcDigitizerTask::CalculatePadResponseCuda(UInt_t padID,
        UInt_t rowID, Float_t x, Float_t y, Int_t *cudaFNumOfPadsInRow) {
752
753     Float_t padW, padH;
754     if (rowID < cudaNInRows) {
755       padW = cudaPwIn;
756       padH = cudaPhIn;
757     } else {
758       padW = cudaPwOut;
759       padH = cudaPhOut;
760     }
761     Float_t padX = padW * ((Float_t) padID - (Float_t) cudaFNumOfPadsInRow[rowID]
          + 0.5); // x-coordinate of pad center
762     Float_t padY = padH * ((Float_t) rowID + 0.5); // y-coordinate of pad center
763
764     Float_t maxX = x - (padX - padW / 2);
765     Float_t minX = x - (padX + padW / 2);
766     Float_t maxY = y - (padY - padH / 2);
767     Float_t minY = y - (padY + padH / 2);
768
769     Float_t coef = 1 / sqrt(2.0) / cudaFSpread;
770     Float_t i1 = (erf(maxX * coef) - erf(minX * coef)) / 2;
771     Float_t i2 = (erf(maxY * coef) - erf(minY * coef)) / 2;
772
773     return i1 * i2;
774 }
775
776 /*
777  * CPU
778  */
779 /**
780  *
781  */
```

```
782 __host__ void TpcDigitizerTask::DistributorCPU(vector<CudaTpcLorenzVector> &
      electronVect) {
783   UInt_t electronGlobalCount = electronVect.size();
784
785   if (electronGlobalCount == 0) {
786     Error("TpcDigitizerTask::Exec", "No electrons at all!!!");
787     return;
788   }
789   if (fPrintDebugInfo) {
790     printf("Using CPU version of algorithm\n");
791     printf("Number of electrons is %u\n", electronGlobalCount);
792   }
793   vector<UInt_t> lightedPadsOut;
794   vector<UInt_t> lightedRowsOut;
795   vector<Float_t> ampsOut;
796   Float_t ampNorm;
797   UInt_t curSectID;
798   UInt_t curTimeID;
799   srand(time(0));
800
801   CudaTpcLorenzVector electronPos;
802   CudaTpcLorenzVector diffuse;
803   Float_t driftl;
804   //UInt_t maxLightedPadsCount = 4.0 * max(Ceil(pwIn / fRadius) * Ceil(phIn /
        fRadius), Ceil(pwOut / fRadius) * Ceil(phOut / fRadius));
805   const Float_t phiStep = TwoPi() / nSectors * 2;
806   for (int i = 0; i < electronGlobalCount; ++i) {
807     lightedPadsOut.resize(0); //lightedPadsOut.clear();
808     lightedRowsOut.resize(0); //lightedRowsOut.clear();
809     ampsOut.resize(0);        //ampsOut.clear();
810     ampNorm = 0;
811     curSectID = 0;
812     curTimeID = 0;
813     if (fPrintDebugInfo) {
814       if (i % 100000 == 0) printf("%u%% of TPC points processed\n", UInt_t(i *
            100.0 / electronGlobalCount));
815     }
816     electronPos = electronVect[i];
817     driftl = zCathode - Abs(electronPos.Z); // length for drifting
```

```
818
819       //attachment
820       if (fAttach) {
821         if (Exp(-driftl * fGas->k()) < gRandom->Uniform()) continue;
822       }
823       //diffusion
824       diffuse = (CudaTpcLorenzVector) {0, 0, 0, 0};
825       if (fDiffuse) {
826         const Float_t sqrtDrift = Sqrt(driftl);
827         const Float_t sigmat = fGas->Dt() * sqrtDrift;
828         const Float_t sigmal = fGas->Dl() * sqrtDrift;
829         diffuse.X = gRandom->Gaus(0, sigmat);
830         diffuse.Y = gRandom->Gaus(0, sigmat);
831         diffuse.Z = gRandom->Gaus(0, sigmal);
832       }
833       //drift distortions
834       if (fDistort) {
835         // TODO: to be implemented
836       }
837       electronPos.X += diffuse.X;
838       electronPos.Y += diffuse.Y;
839       electronPos.Z += diffuse.Z;
840
841       curTimeID = (UInt_t) ((Abs(electronPos.Z) - electronPos.T * fGas->VDrift())
              * nTimeBackets / zCathode);
842       if (curTimeID >= nTimeBackets) continue;
843
844
845
846       Float_t globPhi = ATan2(electronPos.Y, electronPos.X); //angle in global
              coordinates
847       if (globPhi < 0) globPhi += TwoPi();
848       curSectID = (UInt_t) (globPhi / phiStep + 0.5); //index of current sector
849       if (curSectID == nSectors / 2) curSectID = 0;
850       const Float_t sectPhi = curSectID * phiStep;
851       if (electronPos.Z < 0.0) curSectID += (nSectors / 2);
852
853       //local coordinates of electron (sector coordinates)
```

```
854       Float_t localX = -electronPos.X * Sin(sectPhi) + electronPos.Y * Cos(sectPhi
              );
855       Float_t localY = electronPos.X * Cos(sectPhi) + electronPos.Y * Sin(sectPhi)
              - r_min;
856
857       GetAreaCPU(localX, localY, lightedPadsOut, lightedRowsOut, ampsOut, ampNorm)
              ;
858
859       for (UInt_t i = 0; i < ampsOut.size(); ++i) {
860         if ((ampsOut[i] > 0) && (ampNorm > 0)) {
861           fDigitsArray[curSectID][lightedRowsOut[i]][lightedPadsOut[i]][curTimeID]
                  += (ampsOut[i] * ampNorm);
862         }
863
864       }
865     }
866   }
867
868   __host__ void TpcDigitizerTask::GetAreaCPU(Float_t xEll, Float_t yEll, vector<
          UInt_t> &padIDs, vector<UInt_t> &rowIDs, vector<Float_t> &amps,
869       Float_t &ampNorm) {
870     Float_t padW = 0.0, padH = 0.0;
871     Float_t y, x;
872     UInt_t pad = 0, row = 0;
873     Float_t amplithude;
874     Float_t ampSum = 0.0;
875
876     if (Abs(yEll - nInRows * phIn) < fRadius) { //layer with both types of pads (
            inner and outer)
877       y = yEll - fRadius;
878       x = xEll - fRadius - pwIn;
879       do {
880         x += pwIn;
881         row = nInRows - 1; //last inner row
882         if (x > 0.0) {
883           pad = Int_t(fNumOfPadsInRow[row] + Floor(x / pwIn));
884         } else {
885           pad = Int_t(fNumOfPadsInRow[row] - 1 + Ceil(x / pwIn));
886         }
```

```
887         if (pad >= fNumOfPadsInRow[row] * 2) {
888           continue;
889         }
890         /*padIDs[padsCount] = pad;
891          rowIDs[padsCount] = row;
892          (padsCount)++;*/
893         amplithude = CalculatePadResponseCPU(pad, row, xEll, yEll);
894         ampSum += amplithude;
895         amps.push_back(amplithude);
896         padIDs.push_back(pad);
897         rowIDs.push_back(row);
898       } while (x < xEll + fRadius);
899       y = yEll + fRadius;
900       x = xEll - fRadius - pwOut;
901       do {
902         x += pwOut;
903         row = nInRows; //first outer row
904         if (x > 0.0) {
905           pad = Int_t(fNumOfPadsInRow[row] + Floor(x / pwOut));
906         } else {
907           pad = Int_t(fNumOfPadsInRow[row] - 1 + Ceil(x / pwOut));
908         }
909         if (pad >= fNumOfPadsInRow[row] * 2) {
910           continue;
911         }
912         /*padIDs[padsCount] = pad;
913          rowIDs[padsCount] = row;
914          (padsCount)++;*/
915         amplithude = CalculatePadResponseCPU(pad, row, xEll, yEll);
916         ampSum += amplithude;
917         amps.push_back(amplithude);
918         padIDs.push_back(pad);
919         rowIDs.push_back(row);
920       } while (x < xEll + fRadius);
921     } else {
922       if (yEll + fRadius < nInRows * phIn) { // inner pads
923         padW = pwIn;
924         padH = phIn;
925       } else if (yEll - fRadius > nInRows * phIn) { //outer pads
```

```
926           padW = pwOut;
927           padH = phOut;
928        }
929        x = xEll - fRadius - padW;
930        do {
931           x += padW;
932           y = yEll - fRadius - padH;
933           do {
934              y += padH;
935              row = (UInt_t) (y / padH);
936              if (x > 0.0) {
937                 pad = Int_t(fNumOfPadsInRow[row] + Floor(x / padW));
938              } else {
939                 pad = Int_t(fNumOfPadsInRow[row] - 1 + Ceil(x / padW));
940              }
941              if (row >= nRows || pad >= fNumOfPadsInRow[row] * 2) {
942                 continue;
943              }
944              /*padIDs[padsCount] = pad;
945               rowIDs[padsCount] = row;
946               (padsCount)++;*/
947              amplithude = CalculatePadResponseCPU(pad, row, xEll, yEll);
948              ampSum += amplithude;
949              amps.push_back(amplithude);
950              padIDs.push_back(pad);
951              rowIDs.push_back(row);
952           } while (y < yEll + fRadius);
953        } while (x < xEll + fRadius);
954     }
955     if (ampSum != 0) {
956        ampNorm = fGain / ampSum;
957     } else {
958        ampNorm = -1;
959     }
960  }
961
962  __host__ Float_t TpcDigitizerTask::CalculatePadResponseCPU(UInt_t padID, UInt_t
        rowID, Float_t x, Float_t y) {
963     Float_t padW, padH;
```

```
964     if (rowID < nInRows) {
965       padW = pwIn;
966       padH = phIn;
967     } else {
968       padW = pwOut;
969       padH = phOut;
970     }
971 
972     const Float_t padX = padW * ((Float_t) padID - (Float_t) fNumOfPadsInRow[rowID
        ] + 0.5); // x-coordinate of pad center
973     const Float_t padY = padH * ((Float_t) rowID + 0.5); // y-coordinate of pad
        center
974 
975     const Float_t maxX = x - (padX - padW / 2);
976     const Float_t minX = x - (padX + padW / 2);
977     const Float_t maxY = y - (padY - padH / 2);
978     const Float_t minY = y - (padY + padH / 2);
979 
980     const Float_t coef = 1 / Sqrt(2.0) / fSpread;
981     const Float_t i1 = (Erf(maxX * coef) - Erf(minX * coef)) / 2;
982     const Float_t i2 = (Erf(maxY * coef) - Erf(minY * coef)) / 2;
983 
984     return i1 * i2;
985 }
986 
987 Bool_t TpcDigitizerTask::isSubtrackInInwards(const TpcPoint *p1, const TpcPoint
      *p2) { //WHAT AM I DOING???
988     const Float_t x1 = p1->GetX();
989     const Float_t x2 = p2->GetX();
990     const Float_t y1 = p1->GetY();
991     const Float_t y2 = p2->GetY();
992     const Float_t a = (y1 - y2) / (x1 - x2);
993     const Float_t b = (y1 * x2 - x1 * y2) / (x2 - x1);
994     const Float_t minR = fabs(b) / sqrt(a * a + 1);
995 
996     if (minR < r_min) //then check if minimal distance is between our points
997         {
998       const Float_t x = -a * b / (a * a + 1);
999       const Float_t y = b / (a * a + 1);
```

```
1000        if ((x1 - x) * (x2 - x) < 0 && (y1 - y) * (y2 - y) < 0) {
1001          return kTRUE;
1002        }
1003      }
1004      return kFALSE;
1005    }
1006
1007    void TpcDigitizerTask::Finish() {
1008      if (fMakeQA) {
1009        toDirectory("QA/TPC");
1010        Float_t digit = 0.0;
1011        UInt_t iPad_shifted = 0; //needed for correct drawing of fDigitsArray
1012
1013        for (UInt_t iSect = 0; iSect < nSectors; iSect++) {
1014          for (UInt_t iRows = 0; iRows < nRows; ++iRows) {
1015            for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[iRows] * 2; ++iPads) {
1016              iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[
                      iRows];
1017              for (UInt_t iTime = 0; iTime < nTimeBackets; ++iTime) {
1018                digit = fDigitsArray[iSect][iRows][iPads][iTime];
1019                fHisto->_hXY_dig->Fill(iPad_shifted, iRows, digit);
1020                fHisto->_hSect_dig->Fill(iSect, digit);
1021                fHisto->_hX_dig->Fill(iPad_shifted, digit);
1022                fHisto->_hY_dig->Fill(iRows, digit);
1023                fHisto->_hZ_dig->Fill(iTime, digit);
1024                fHisto->_h3D_dig->Fill(iPad_shifted, iRows, iTime, digit);
1025                if (digit > 0.0) fHisto->_hADC_dig->Fill(digit);
1026              }
1027            }
1028          }
1029        }
1030
1031        for (UInt_t iRows = 0; iRows < nRows; ++iRows) {
1032          for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[iRows] * 2; ++iPads) {
1033            iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[
                    iRows];
1034            for (UInt_t iTime = 0; iTime < nTimeBackets; ++iTime) {
1035              digit = fDigitsArray[0][iRows][iPads][iTime];
1036              //pad activity
```

```
1037             //if (digit > 1000.0) {
1038             //      fHisto->_hXY_dig->Fill(iPad_shifted, iRows, 1.0);
1039             //}
1040             //                          fHisto->_hXY_dig->Fill(iPad_shifted, iRows,
                   digit);
1041             fHisto->_h3D_dig->Fill(iPad_shifted, iRows, iTime, digit);
1042           }
1043         }
1044       }
1045
1046       UInt_t sec = 3;
1047       for (UInt_t iTime = 0; iTime < nTimeBackets; ++iTime) {
1048         for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[1] * 2; ++iPads) {
1049           iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[1];
1050           digit = fDigitsArray[sec][1][iPads][iTime];
1051           fHisto->_hXT_dig_1->Fill(iPad_shifted, iTime, digit);
1052         }
1053         for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[5] * 2; ++iPads) {
1054           iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[5];
1055           digit = fDigitsArray[sec][5][iPads][iTime];
1056           fHisto->_hXT_dig_5->Fill(iPad_shifted, iTime, digit);
1057         }
1058         for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[10] * 2; ++iPads) {
1059           iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[10];
1060           digit = fDigitsArray[sec][10][iPads][iTime];
1061           fHisto->_hXT_dig_10->Fill(iPad_shifted, iTime, digit);
1062         }
1063         for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[20] * 2; ++iPads) {
1064           iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[20];
1065           digit = fDigitsArray[sec][20][iPads][iTime];
1066           fHisto->_hXT_dig_20->Fill(iPad_shifted, iTime, digit);
1067         }
1068         for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[40] * 2; ++iPads) {
1069           iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] - fNumOfPadsInRow[40];
1070           digit = fDigitsArray[sec][40][iPads][iTime];
1071           fHisto->_hXT_dig_40->Fill(iPad_shifted, iTime, digit);
1072         }
1073         //             for (UInt_t iPads = 0; iPads < fNumOfPadsInRow[60] * 2; ++
               iPads) {
```

```
1074          //                    iPad_shifted = iPads + fNumOfPadsInRow[nRows - 1] -
                  fNumOfPadsInRow[60];
1075          //                    digit = fDigitsArray[sec][60][iPads][iTime];
1076          //                    fHisto->_hXT_dig_60->Fill(iPad_shifted, iTime, digit);
1077          //                }
1078        }
1079
1080        fHisto->Write();
1081        gFile->cd();
1082      }
1083    }
1084
1085    ClassImp(TpcDigitizerTask)
```

C Исходный код файла cuda_TMath.patch

cuda_TMath.patch

```
--- ./install/include/root/TMath.h.old  2013-05-17 05:53:32.440623924 +0400
+++ ./install/include/root/TMath.h  2013-05-17 12:52:02.413353486 +0400
@@ -502,7 +502,7 @@
 #endif

 inline Int_t TMath::IsNaN(Double_t x)
-#if (defined(R__ANSISTREAM) || (defined(R__MACOSX) && defined(__arm__))) && !
    defined(_AIX)
+#if (defined(R__ANSISTREAM) || (defined(R__MACOSX) && defined(__arm__))) && !
    defined(_AIX) && !defined(__CUDACC__)
 #if defined(isnan) || defined(R__SOLARIS_CC50) || defined(__INTEL_COMPILER)
       // from math.h
     { return ::isnan(x); }
```

D Исходный код файла cuda_CMake.patch

cuda_ CMake.patch

```
1  --- CMakeLists_old.txt  2013-05-29 19:56:48.000000000 +0400
2  +++ CMakeLists.txt  2013-06-14 10:57:26.637421634 +0400
3  @@ -16,7 +16,40 @@
4    ${CMAKE_SOURCE_DIR}/lhetrack
5    ${CMAKE_SOURCE_DIR}/kalman
6    )
7  -
8  +###########
9  +set(TpcDigitizerTask_SRCS
10 + TpcDigitizerTask.cu
11 + TpcDigitizerTask.h
12 + )
13 +if(CUDA_VERSION)
14 +     CUDA_INCLUDE_DIRECTORIES(
15 +    ${INCLUDE_DIRECTORIES}
16 +    ${CMAKE_CURRENT_SOURCE_DIR}
17 +    /opt/cuda/include/
18 +    )
19 +
20 +     add_definitions(-DMULTIPLIER=2)
21 +     set(BUILD_SHARED_LIBS ON)
22 +     set(CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE ON)
23 +     list(APPEND CUDA_NVCC_FLAGS  -arch=sm_20 -Xcompiler -fPIC) #flag fPIC for
     gcc for library generate, w - for delete warnings output(DELETE HIM), -arch:
     cuda architerture
24 +     CUDA_ADD_LIBRARY(TpcDigitizerTask
25 + ${TpcDigitizerTask_SRCS}
26 + SHARED
27 + #STATIC
28 +     )
29 +     CUDA_BUILD_CLEAN_TARGET()
30 +else()
31 + set(TPC_LINKDEF  tpcLinkDef.h)
32 + set(TPC_DICTIONARY ${CMAKE_CURRENT_BINARY_DIR}/tpcDict.cxx)
```

```
33 + #ROOT_GENERATE_DICTIONARY("${TPC_HEADERS}" "${TPC_LINKDEF}" "${TPC_DICTIONARY}
     " "${INCLUDE_DIRECTORIES}")
34 + #SET(TPC_SRCS ${TPC_SRCS} ${TPC_DICTIONARY} dbgstream.cxx)
35 + SET(TpcDigitizerTask_SRCS ${TpcDigitizerTask_SRCS} ${TPC_DICTIONARY} )
36 + add_library(TpcDigitizerTask SHARED ${TpcDigitizerTask_SRCS})
37 + target_link_libraries(TpcDigitizerTask ${ROOT_LIBRARIES})
38 + set_target_properties(TpcDigitizerTask PROPERTIES VERSION 0.0.0 SOVERSION 0 )
39 + install(TARGETS TpcDigitizerTask DESTINATION ${CMAKE_BINARY_DIR}/lib)
40 +endif()
41 +###########
42  include_directories( ${INCLUDE_DIRECTORIES})
43 
44  set(LINK_DIRECTORIES
45 @@ -114,7 +147,7 @@
46  MpdParticleIdentification.cxx
47  TpcClearerTask.cxx
48  TpcDistributor.cxx
49 -TpcDigitizerTask.cxx
50 +#TpcDigitizerTask.cxx
51  TpcDigitizerQAHistograms.cxx
52  Tpc2dCluster.cxx
53  #TpcSingleDistributor.cxx
54 @@ -137,7 +170,7 @@
55  SET(TPC_SRCS ${TPC_SRCS} ${TPC_DICTIONARY} )
56 
57  add_library(tpc SHARED ${TPC_SRCS})
58 -target_link_libraries(tpc ${ROOT_LIBRARIES})
59 +target_link_libraries(tpc ${ROOT_LIBRARIES} "${CMAKE_BINARY_DIR}/lib/
     libTpcDigitizerTask.so")
60  set_target_properties(tpc PROPERTIES VERSION 0.0.1 SOVERSION 0 )
61 
62  ################ install ###################
```