Нижегородский государственный университет им. Н.И. Лобачевского
Национальный исследовательский университет
Программа повышение конкурентоспособности ННГУ им. Н.И. Лобачевского
Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений»

Основная образовательная программа
011800 Радиофизика
квалификация (степень) магистр
Учебно-методическая разработка по дисциплине
«Моделирование сигнальных процессов в нейронных сетях мозга»

Есир П. М., Симонов А. Ю.

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ GPGPU ВЫЧИСЛЕНИЯ ДЛЯ МОДЕЛИРОВАНИЯ ДИНАМИКИ СПАЙКОВЫХ НЕЙРОННЫХ СЕТЕЙ

Нижний Новгород
2014 год

УДК 519.876.5, 004.942, 51-76, 591.181, 51-37
ББК Е903.9
Е83

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ GPGPU ВЫЧИСЛЕНИЯ ДЛЯ МОДЕЛИРОВАНИЯ ДИНАМИКИ СПАЙКОВЫЙ НЕЙРОННЫХ СЕТЕЙ.
Есир П.М., Симонов А.Ю. Учебно-методическое пособие. - Нижний Новгород: Нижегородский госуниверситет, 2014. 82

Рецензент: к.ф.-м.н., **А.С. Пимашкин**

В учебно-методическом пособии описаны приёмы написания параллельных приложений для моделирования динамики спайковых нейронных сетей с применением технологии GPGPU (General-purpose computing on graphics processing units — вычисления общего назначения на графических процессорах). А также приёмы использования готового программного продукта NNSim для проведения GPGPU вычислений. В пособии уделено внимание краткому описанию физиологических механизмов генерации и передачи потенциалов действия (спайков) в живых нейронных сетях и математическим моделям этих процессов.

Также приведены примеры использования программы-симулятора NNSim. Подробно разобран механизм его работы, рассмотрено каким образом достигается ускорение вычислений и приведён список функций, реализующих все возможности симулятора. Учебно-методическое пособие также содержит ряд примеров использования программы-симулятора. Приведены достигнутые уровни ускорения для сетей разного размера. Симулятор написан на основе свободных программных продуктов и размещён в открытом доступе вместе с исходным кодом.

Пособие предназначено для студентов 1 курса магистратуры, обучающихся по направлению подготовки "011800 Радиофизика", по профилю "Нелинейные колебания и волны" в рамках проведения практических работ по курсу "Моделирование сигнальных процессов в нейронных сетях мозга". Также пособие может быть использовано студентами старших курсов, обучающихся по направлению подготовки 010300 «Фундаментальная информатика и информационные технологии» и занимающимися математическим моделированием динамических процессов в нейронных системах мозга.

Ответственный за выпуск:
председатель методической комиссии _____ факультета ННГУ, _____

Оглавление

Модельный подход в нейронауке

Динамику мембранного потенциала нейрона можно описать математически, впервые это было сделано Аланом Ходжкиным и Эндрю Хаксли. В своих пионерских работах по исследованию проводимости гигантского аксона кальмара [1] ими было экспериментально показано, что проводимость мембраны нейрона подчиняются определённым закономерностям, которые они записали в виде дифференциальных уравнений. Решая эти уравнения, можно воспроизвести эффекты, возникающие при исследовании живых нейронов или даже предсказать новые. Со времён работ Ходжкина и Хаксли появилось множество новых моделей нейронов и сетей нейронов, как простых так достаточно детальных в которых воспроизводятся те или иные значимые эффекты присущие живым нейронам. Результаты этих исследований вылились в отдельную отрасль нейронауки, называемую вычислительной нейронаукой. Поскольку эта область науки сосредоточена на составлении и исследовании дифференциальных уравнений описывающих колебательно-волновые процессы связанные с генерацией нервных импульсов богатый аппарат теории колебаний и динамических систем созданный для решения радиофизических задач. Кроме того, задачи включающие в себя моделирование динамических систем состоящих из множество элементов, а при всё более детальном исследовании нейронных сетей мозга увеличение числа элементов неизбежно, требует больших вычислительных мощностей. Поэтому высокопроизводительные вычисления и в частности GPGPU (General-purpose computing on graphics processing units — вычисления общего назначения для графических процессоров) оказались большим подспорьем для вычислительной нейронауки. Следует отметить, что численное моделирование имеет ряд преимуществ перед проведением натуральных экспериментов: их проведение значительно проще чем экспериментов *in vitro* с нейронными культурами (это культура из нервных клеток выращиваемая в специальной среде на мультиэлектродной матрице с помощью которой можно детектировать их

электрическую активность) и *in vivo* экспериментов на целом мозге. Кроме того численное моделирование даёт невиданные возможности автоматизации, прозрачности и воспроизводимости которые не присущи натуральным экспериментам. Перечисленные выше преимущества вместе с наличием богатого теоретического базиса для анализа дифференциальных уравнений задающих динамику нейронов, а также стремительное развитие технологий высокопроизводительных вычислений делают вычислительную нейронауку весьма привлекательной и плодотворной областью науки.

Биологические нейроны и синапсы

Типичный нейрон состоит из 3-х основных частей: тела (сомы), дендритов и аксона, Рисунок 1.

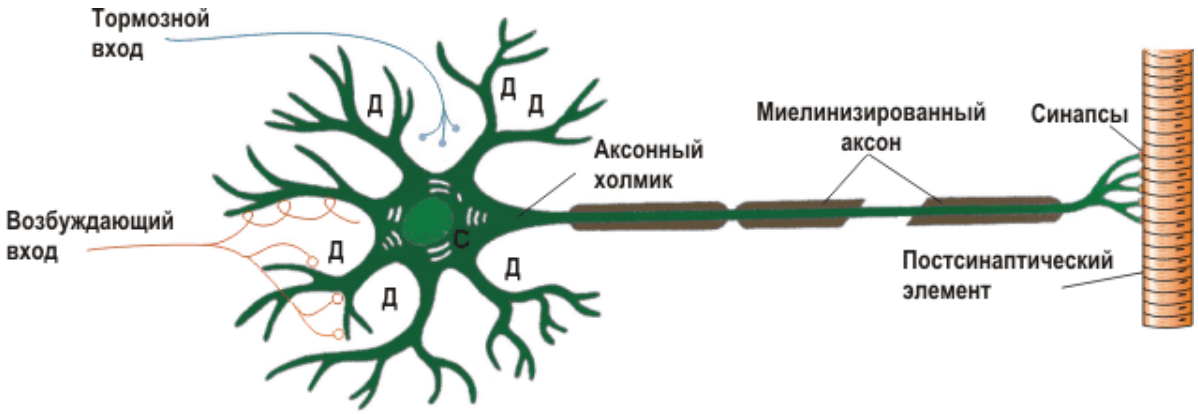

Рисунок 1 — Схематичное идеализированное изображение нейрона. *(Адаптировано из Атлас по физиологии. В двух томах. Том 1: учеб. пособие / А. Г. Камкин, И. С. Киселева - 2010. - 408 с.)*

Дендриты принимают нервные импульсы от других нейронов, далее они суммируются в теле и если суммарный сигнал достаточно сильный, чтобы возбудить нейрон, то волна возбуждения в виде потенциала действия (по другому иногда называемая - спайком) распространяется дальше по аксону и передаётся другим нейронам. Тело клетки окружено липидной оболочкой, являющейся хорошим изолятором: она не пропускает ни электроны, ни ионы. Ионные составы цитоплазмы нейрона и межклеточной жидкости различаются.

В цитоплазме концентрация ионов калия выше, а концентрация натрия и хлора ниже, в межклеточной жидкости наоборот. Это связано с работой ионных насосов расположенных на поверхности нейрона, они представляющих из себя белковые структуры которые встроены в липидную мембрану. Ионные насосы постоянно перекачивают определённые типы ионов против градиента концентрации. Самым известным и изученным из таких насосов является натрий-калиевый насос: он выводит 3 иона натрия наружу, а внутрь нейрона забирает 2 иона калия. На рисунке 2 изображён ионный состав нейрона и отмечены ионные насосы. Благодаря работе этих насосов в нейроне образуется

равновесная разность потенциалов между внутренней стороной мембраны, заряженной отрицательно, и внешней, заряженной положительно.

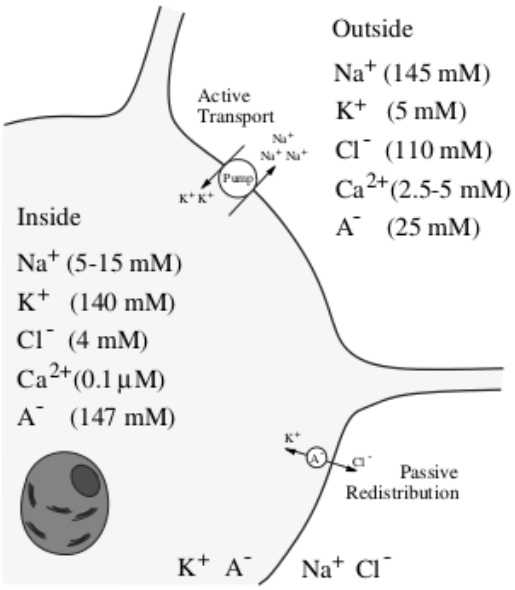

Рисунок 2 — Ионный состав нейрона. Адаптировано из E.M. Из *Izhikevich, Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting, USA, MA, Cambridge: The MIT Press., 2007*

Кроме ионных насосов на поверхности нейрона есть ещё и ионные каналы, они также представляют из себя белковые комплексы, встроенные в липидную оболочку и образующие поры через которые могут течь определённые ионы. При изменении разности потенциалов между внешней и внутренней стороной мембраны или при химическом воздействии на эти каналы определённых веществ – нейромедиаторов они могут открываться или закрываться, тем самым увеличивая или уменьшая токи определённого типа ионов. Ионные каналы чувствительные к химическому воздействию обычно называют рецепторами. В типичном нейрона двумя основными типами каналов определяющими механизм генерации потенциала действия (спайка), являются калиевые и натриевые ионные каналы, которые в зависимости от мембранного потенциала могу пропускать ионы калия или натрия соответственно.

Упрощённая схема генерации спайка следующая. Если мембранный потенциал нейрона превышает некоторый порог, например из-за внешнего

воздействия на него, то открываются натриевые каналы, а так как снаружи нейрона больше натрия, то возникает электрический ток направленный внутрь нейрона, что ещё больше увеличивает мембранный потенциал и ещё сильнее открывает натриевые каналы, возникает положительная обратная связь, происходит резкое увеличение мембранного потенциала и генерация спайка. Но начиная с какого-то значения потенциала, более высокого чем пороговый потенциал открытия натриевых каналов, открываются и калиевые каналы, благодаря чему ионы калия начинают течь в наружу уменьшая мембранный потенциал, тем самым возвращая его к равновесному значению, после чего уже натрий-калиевый насос возвращает уже ионные концентрации в исходное состояние. Если же первоначальное возбуждение меньше порога открытия натриевых каналов, то нейрон вернётся к своему равновесному состоянию без генерации спайка. Что интересно, амплитуда генерируемого спайка слабо зависит от амплитуды возбуждающего тока, либо спайк есть, либо его нет, закон «все или ничего», рисунок 3.

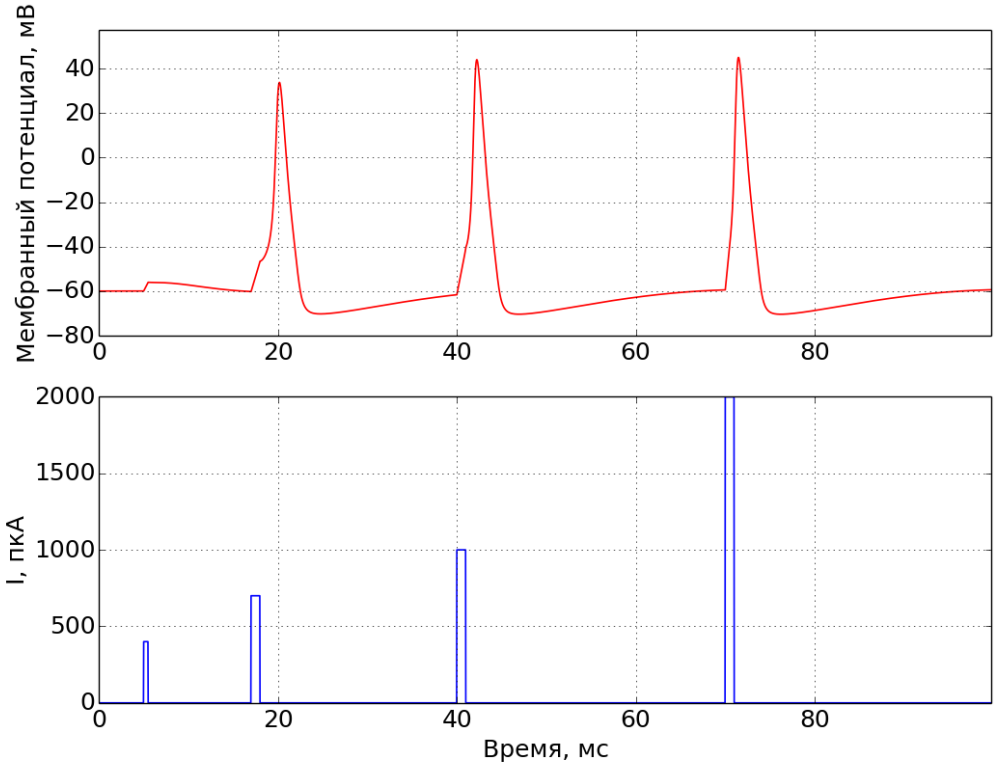

Рисунок 3 — Генерация потенциала действия (спайка) в модели Ходжкина-Хаксли. Внизу изображён ток подаваемый по направлению вовнутрь нейрона, а вверху мембранный потенциал нейрона.

Взаимодействие нейронов друг с другом происходит с помощью синапсов, рисунок 4, образующихся в местах контакта аксона (терминали аксона) одного нейрона с дендритами другого. Нейрон, от которого идёт спайк, называется пресинаптическим, а тот к которому идёт спайк постсинаптическим. При возникновении спайка на пресинаптическом нейроне в нём открываются кальциевые ионные каналы, при попадании которого внутрь терминали происходит разрушение оболочки синаптических пузырьков (везикул, см Рисунок 4) и они сливаются с мембранной пресинапса благодаря чему происходит выделение нейромедиатора в синаптическую щель. В случае если это возбуждающий нейрон, он выделят возбуждающий нейромедиатор, обычно, глутамат под воздействием которого на постсинаптическом нейроне открываются AMPA-рецепторы которые начинают пропускать натриевые ионы внутрь постсинаптического нейрона, а дальше происходит цепь событий, приводящих к генерации спайка, уже описанных выше. После того как нейромедиатор прореагировал с рецептором он разрушается ферментами, а впоследствии вновь синтезируется в терминали, либо непосредственно возвращается в пресинаптическую терминаль путём обратного захвата. Кинетику этого процесса также можно описать математически, о чём будет сказано далее.

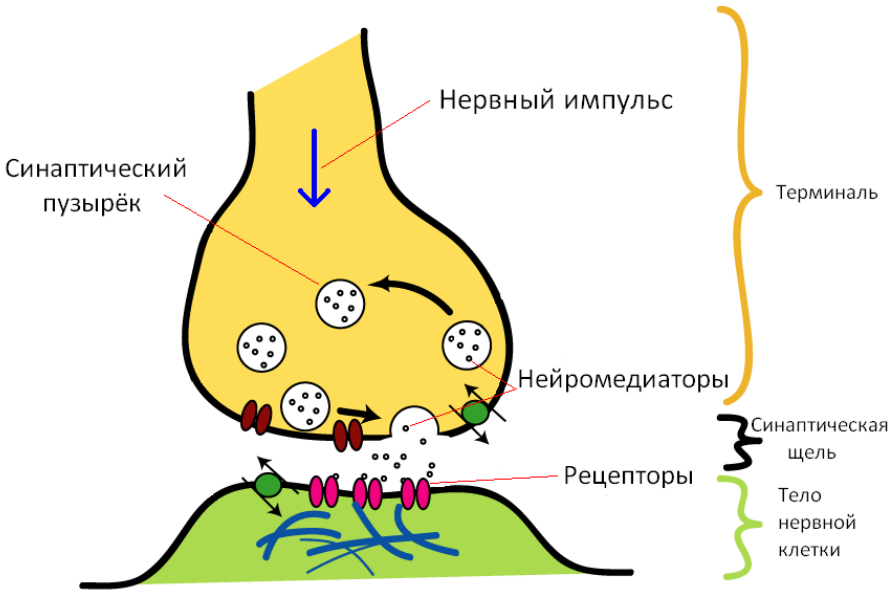

Рисунок 4 — Схематичное изображение синапса (*из ru.wikipedia.org)*

Кроме возбуждения нейроны могут и тормозить друг друга. В случае если пресинаптический нейрон тормозный, то он выделяет в синаптическую щель тормозный нейромедиатор - ГАМК (по англ. GABA) который способствует открытию GABA-рецепторов, пропускающих ионы хлора. А так как снаружи нейрона хлора больше, то хлор диффундирует внутрь нейрона, из-за чего отрицательный заряд на внутренней стороне мембраны увеличивается (ионы хлора заряжены отрицательно), вгоняя нейрон в ещё более неактивное состояние. В таком состоянии нейрон труднее возбудить.

Модель нейрона Ижикевича

Существует большое количество математических моделей позволяющих воспроизводить процессы, описанные в предыдущем параграфе, среди всех этих моделей одной из самых распространённых является — модель Ижикевича [2]. Она отличается своей математической простотой, благодаря чему численные расчёты с её использованием довольно эффективны. Вместе с тем при всей своей простоте она позволяет воспроизводить богатый набор динамических режимов характерных для живых нейронов. Уравнение для этой модели:

$$\begin{gathered} C_m \frac{dV}{dt} = k(V - V_r)(V - V_t) - U + I_{syn} + I_{ex} \\ \frac{dU}{dt} = a(b(V - V_r) - U) \\ if V > V_{peak}, then \\ V = c \\ U = U + d \end{gathered} \qquad (1)$$

где C_m — ёмкость нейрона пкФ, V — мембранный потенциал мВ, V_{peak} — пиковое значение мембранного потенциала при достижении которого происходит генерация спайка и сбрасывание значений V и u, V_r, V_t — это вспомогательные параметры имеющие размерность напряжения, c — значение мембранного потенциала к которому сбрасывается состояние нейрона при возникновении спайка, U — вспомогательная переменная. I_{syn} и I_{ex} — суммарный синаптический ток и постоянный внешний приложенный ток, соответственно, пкА, a, b, d, k — вспомогательные параметры.

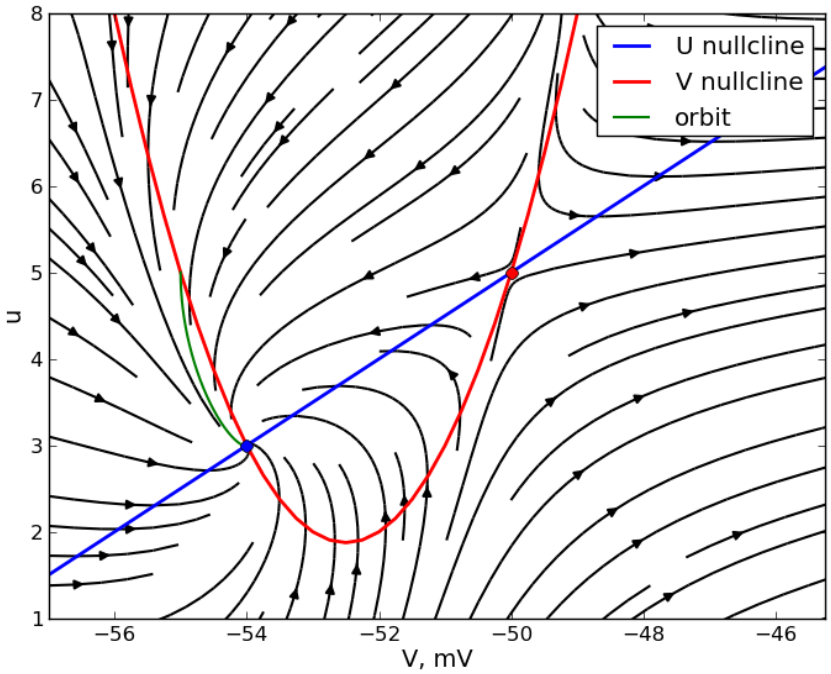

Рисунок 5 — Фазовый портрет модели Ижикевича в случае отсутствия внешнего тока, нейрон находиться в возбудимом режиме. Синяя линия — изоклина вертикальных касательных, красная — горизонтальных, зелёным обозначена траектория системы. Синей точкой отмечено устойчивое состояние равновесия — устойчивый узел, а красным неустойчивое седло.

В случае если на нейрон не подан постоянный внешний ток, нейрон не генерирует спайки без внешнего воздействия, рисунок 5 если же воздействие достаточно сильно, чтобы перевести состояние системы правее неустойчивой сепаратрисы, то мембранный потенциал начинает неограниченно возрастать вплоть до пикового значения когда он согласно формуле (1) сбрасывается к значению c. Если же к нейрону приложен достаточно сильный постоянный внешний ток происходит генерация спайков с постоянной частотой, рисунок 6. В этом режиме нульклины не пересекаются, поэтому в системе нет состояний равновесия, поэтому из любых начальных условиях состояние системы начинает уходить в сторону увеличения V рано или поздно пересекая пороговое значение, после чего значения V и u сбрасываются и цикл генерации спайка

повторяется.

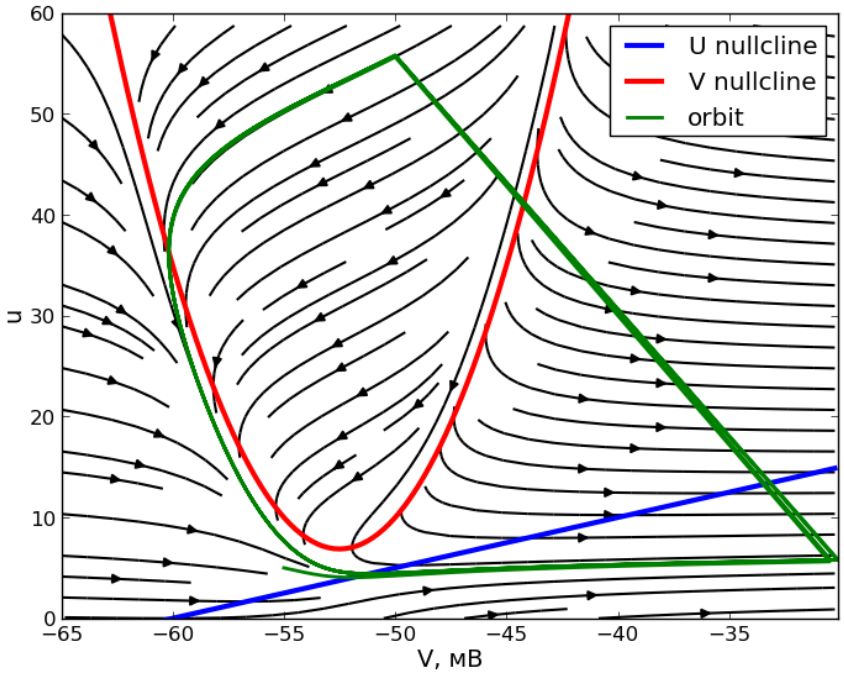

Рисунок 6 — Фазовый портрет модели Ижикевича в случае приложенного внешнего тока, нейрон находиться в автоколебательном режиме.

На рисунке 7 приведены осциллограммы мембранного потенциала одиночного нейрона при различных значениях параметров модели.

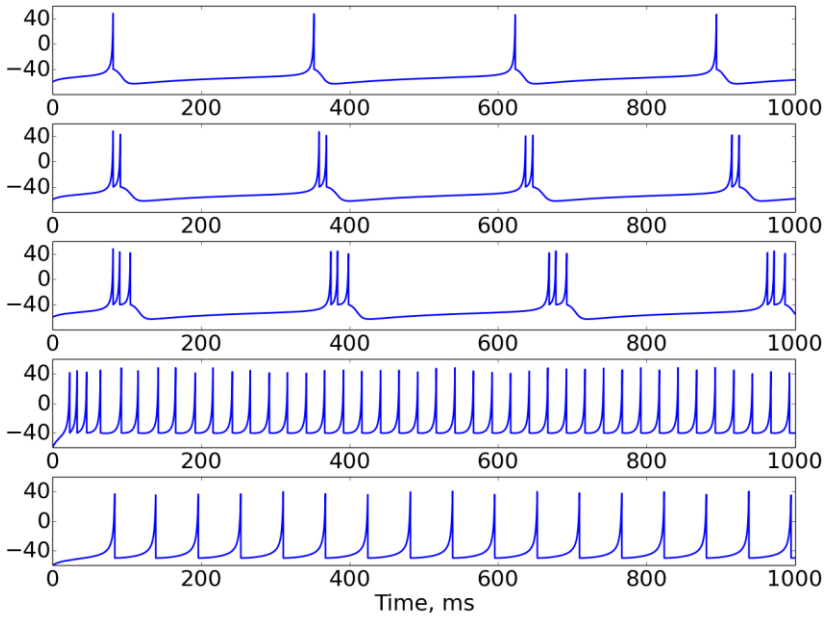

Рисунок 7 — Осциллограммы в модели Ижикевича. Различные режимы соответствуют различным нейронам наблюдаемым в живых нейронных сетях.

Модель синапса Цодыкса-Маркрама

Одной из важных особенностей нейронных сетей является кратковременная синаптическая пластичность. Благодаря этому явлению сила связи между нейронами может уменьшаться либо увеличиваться в зависимости от частоты поступающих на синапс спайков. Биофизические механизмы этого явления не до конца изучены, однако наиболее вероятным является механизм по которому данный эффект связан с истощением нейромедиатора (синаптического ресурса) и с высвобождением кальция из эндоплазматического ретикулума на пресинапсе, в зависимости от того какой процесс доминирует может наблюдаться либо явление депрессии либо фасилитации.

Явление депрессии заключается в уменьшении амплитуды ответа последующего спайка по отношению к амплитуде предыдущего в зависимости от интервала прошедшего после последнего спайка. Фасилитация же — это обратный процесс, при котором амплитуда ответа от последующего спайка больше предыдущего. Оба эти явления могут быть описаны математически в виде модели, одна из первых таких моделей была предложена Михаилом

Цодыксом и Генри Маркрамом в работе [3].

$$\begin{cases} \dot{x} = \frac{1-x}{\tau_{rec}} - u x \delta(t - t_{sp} - d) \\ \dot{u} = \frac{-u}{\tau_{fac}} + U(1-u)\delta(t - t_{sp} - d) \\ \dot{y} = \frac{-y}{\tau_{psc}} + w x u \delta(t - t_{sp} - d) \end{cases} \quad (2)$$

Где x — количество нейромедиатора в пресинаптической терминали, u — количество запасённого кальция в терминали, τ_{rec} — время восстановления нейромедиатора, τ_{fac} — время выводы кальция из пресинаптической терминали - эта переменная определяет усиливающие способности синапса, δ — дельта функция, t, t_{sp} — текущее время и время генерации спайка на пресинапсе, d — задержка от пресинапса к постсинапсу, U — доля нейромедиатора от всего его количества в пресинапсе, которая выделяется за один спайк, y, τ_{psc} — количество нейромедиатора в синаптической щели и характерное время его истечения, в различных моделях она может задавать либо ток на постсинаптическом нейроне (4) (*current based synapse*), либо проводимость (3) (*conductance based synapse*).

В случае если это проводимость то вклад в общий постсинаптический ток данного синапса задаётся уравнением (3)

$$I_{syn} = y(V - V_{rev}) \quad (3)$$

Где I_{syn} синаптический ток пкА, в этом случае w из формулы (2) это вес связи выраженный в нСм, V_{rev} — реверсивный потенциал мВ.

Если y модулирует ток:

$$I_{syn} = y \quad (4)$$

Где I_{syn} синаптический ток пкА, в этом случае w из формулы (2) это вес связи в пкА.

Данная система дифференциальных уравнений может быть легко решена аналитически. При отсутствии спайков она является линейным дифференциальным уравнением с решением в виде экспоненциального

стремления переменной ***x*** к состоянию равновесия в единице с характерным временем τ_{rec}, и экспоненциального спадания переменных ***u***, ***y*** к нулю с характерными временами τ_{fac}, τ_{rec}, рисунок 8. В случае же возникновения спайка происходит скачкообразное изменение обоих переменных, ***x*** уменьшается на ***u*x***, а ***u*** увеличивается на ***U*(1-u)***. Дальше, в следующий временной шаг после спайка динамика является точно такой же как при отсутствии спайков, только начальные условия другие. Данное свойство делает эту модель очень удобной для параллелизации, можно считать на разных потоках динамики каждого отдельного синапса синхронизируя их лишь в моменты спайков. Также ввиду наличия точного аналитического решения этой системы при вычислениях на компьютере можно не использовать алгоритмы интегрирования обыкновенных дифференциальных уравнений, а использовать точечное отображение готового аналитического решения поэтому отпадает необходимость использовать более длинный временной шаг интегрирования.

Значения переменных на каждом временном шаге интегрирования вычисляемое исходя из точного аналитического решения.

$$\begin{cases} x_{ij}^{t+1} = 1 - x_{ij}^{t} e^{\frac{-dt}{\tau_{ij}^{rec}}} \\ u_{ij}^{t+1} = u_{ij}^{t} e^{\frac{-dt}{\tau_{ij}^{fac}}} \\ y_{ij}^{t+1} = y_{ij}^{t} e^{\frac{-dt}{\tau_{ij}^{psc}}} \end{cases} \qquad (5)$$

i, j индексы пресинаптического и постсинаптического нейрона, ***dt*** — временной шаг интегрирования. Значение переменных в случае если в данный момент времени произошёл спайк просто инкрементируются согласно формуле (2) на ***-ux | ux*** для ***x | y***, либо на ***U(1-u)*** для ***u:***

$$\begin{cases} u_{ij}^{t+1} = \bar{u}_{ij}^{t+1} + U_{ij}\left(1 - \bar{u}_{ij}^{t+1}\right) \\ x_{ij}^{t+1} = \bar{x}_{ij}^{t+1} - u_{ij}^{t+1}\bar{x}_{ij}^{t+1} \\ y_{ij}^{t+1} = \bar{y}_{ij}^{t+1} + w_{ij}u_{ij}^{t+1}\bar{x}_{ij}^{t+1} \end{cases} \quad (6)$$

Где $\bar{y}, \bar{u}, \bar{x}$ обозначаются синаптические переменные в текущий момент времени ещё до учёта воздействия спайка из формулы (5). Суммарный постсинаптический ток на нейроне:

$$\left\{ \quad \right. \quad (7)$$

Суммирование ведётся по индексам пресинаптических нейронов i. Существенно можно упростить вычисления если считать, что время затухания постсинаптичского тока τ_{psc} одинаково для всех синапсов одного типа данного нейрона. Например считать, что для одного нейрона все входящие возбуждающие связи имеют $\tau_{psc} = \tau_{psc\_exc}$, а все тормозные $\tau_{psc} = \tau_{psc\_inh}$. В этом случае можно совокупность переменных y для каждого входящего синапса данного нейрона свести к 2-м переменным y_{exc}, y_{inh} которые будут вычисляться для каждого нейрона, а не синапса.

(8)

(9)

В данной работе используется модель синапса базирующаяся на проводимости (*conductance based synapse*), поэтому:

$$y(t) = y(t-\Delta t)\,e^{-\Delta t/\tau_y} \qquad (10)$$

Так как нейронов в крупномасштабных сетях меньше чем синапсов это изрядно сократит количество вычислений. Данный подход не лишает модель сети общности ввиду того, что изменение времени затухания y для синапса в некоторых пределах можно скомпенсировать изменением веса этого синапса. Таким образом в случае необходимо сборки сети для которого разные входящие связи имеют разные времена затухания постсинаптического тока нужно установить для всего нейрона среднее значение времени затухания и подкорректировать связи.

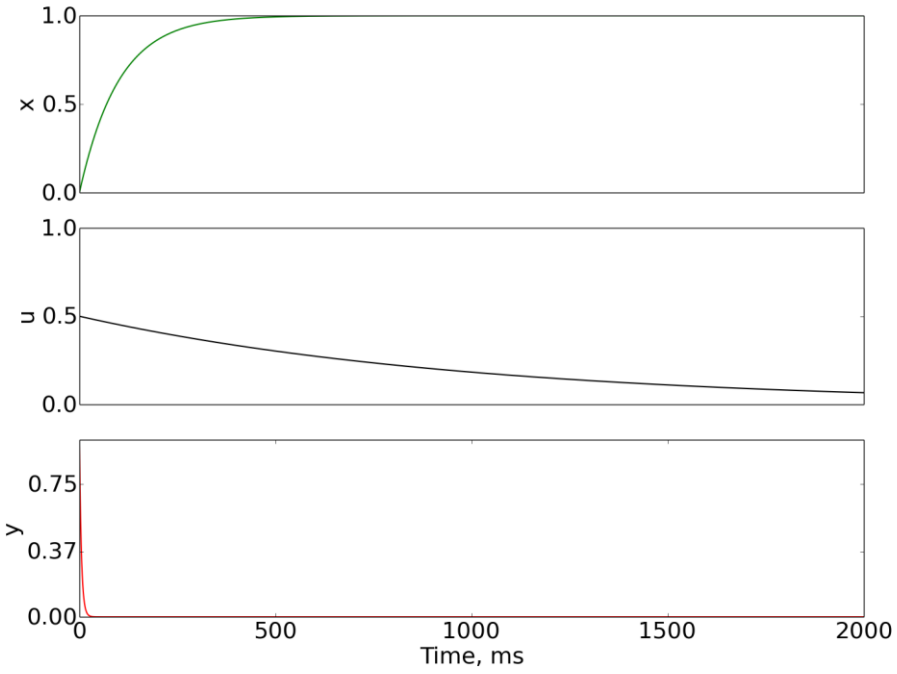

Рисунок 8 — Динамика модели Цодыкса в случае отсутствия спайков.

На рисунках 9 и 10 изображена динамика этой системы в случае если на вход синапса с одинаковой частотой подаётся последовательность спайков, на рисунке 10 модель воспроизводит явление фасилитации, а на рисунке 9 явление

депрессии.

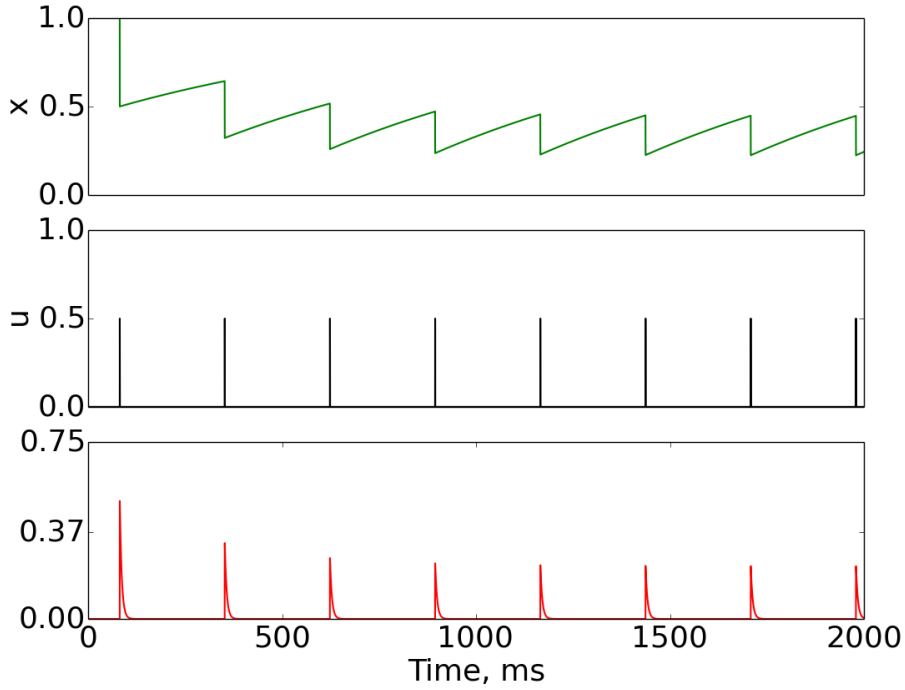

Рисунок 9 — Динамика синаптических переменных в модели Цодыкса в случае депрессии.

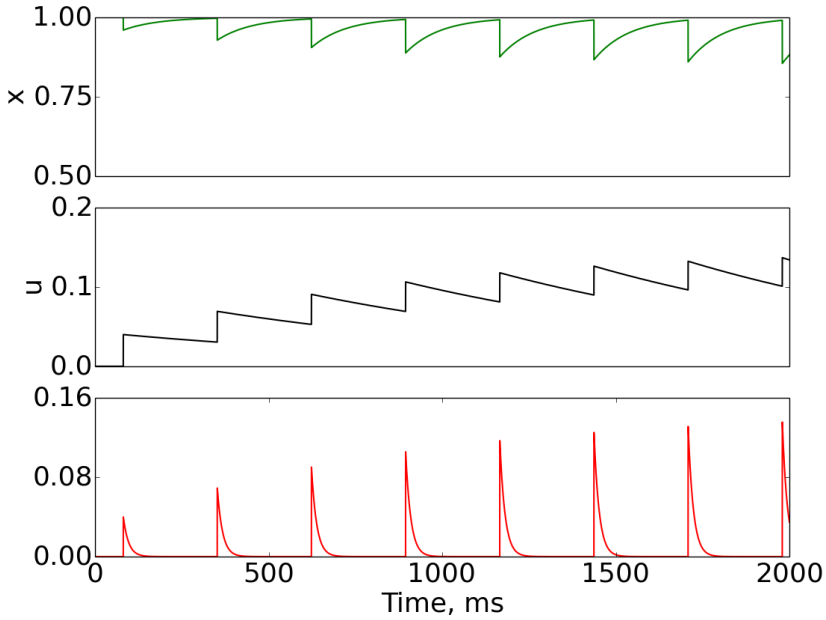

Рисунок 10 — Динамика синаптических параметров в модели Цодыкса в случае фасилитации.

Расчёт динамики спайковых нейронных сетей

Существует большое количество программных продуктов так или иначе решающих численно приведённые выше уравнения для синапсов и нейронов, либо подобные им, некоторые из них: NEURON, NEST, BRIAN, NeMo и др. Подобные программные средства называются симуляторами, потому что они в некотором роде симуляируют процессы происходящие в живых нейронных сетях. Данное методическое пособие не включает в себя описание возможностей этих программных продуктов, поэтому особо не вникая в детали только отметим, почему они не являются достаточно удобными для некоторых задач. Несмотря на ряд возможностей которые предоставляют эти симуляторы, включая возможность проведения параллельных высокопроизводительных вычислений на кластерах, в них нет возможности использовать вычислительные ресурсы графических процессоров. Кроме того несмотря на то, что почти во всех них изначально была заложена возможность написания модулей расширяющих их функционал, этот процесс довольно трудоёмок ввиду их сложного внутреннего устройства, также для некоторых задач они являются недостаточно гибкими. Например симулятор NEST вычисляет значения синаптических переменных x, u лишь в моменты возникновения спайка, из-за чего не представляется возможным узнать их динамику в моменты времени между спайками. Этот недостаток является особенно критичным в случае моделирования популяционных разрядов, при которых лавинообразное нарастание активности с последующей генерацией разряда чередуется с периодами относительно низкой активности, и и в эти моменты времени практически невозможно получить из симулятора динамику синаптических переменных.

Единственный симулятор из вышеуказанных который обладает возможностью использования вычислительных мощностей видеопроцессора — NeMo не включает в себя модель синапса Цодыкса-Маркрама, а следовательно не позволяет моделировать процессы кратковременной пластичности. Поэтому

для решения ряда задач связанных с расчётом динамики относительно крупных спайковых нейронных сетей с пластичными синапсами и чтобы избежать ограничений вышеупомянутых симуляторов, была написана оригинальная программа-симулятор позволяющая как производить параллельные вычисления с использованием ресурсов видеопроцессора, так и моделировать кратковременную синаптическую пластичность с помощью модели Цодыкса-Маркрама с возможность вычисления значений синаптических переменных в межспайковые интервалы.

Для моделирования электрических сигналов в нейронных культурах необходимо моделировать сети состоящие как минимум из нескольких сотен или даже тысяч нейронов, чтобы соответствовать количеству нейронов наблюдаемых в культуре. Что актуализирует необходимость применения технологий параллельных высокопроизводительных вычислений. В компьютерной программе выполняющей расчёт должны быть учтены основные параметры нейронной культуры, такие как количество нейронов, связность сети, синаптические параметры, задержки и пр. Для выполнения этих задач наиболее логичным и простым является создание массивов для каждого нейронного параметра с размерностью равной количеству нейронов, а для синапсов с размерностью равной количеству связей.

После создания массивов со всеми параметрами сети и их наполнения релевантными значениями необходимо численно решить систему которую они задают. Ниже представлен фрагмент исходного кода программы на языке C++ реализующий один из вариантов решения этой задачи.

```
float izhik_Vm(float Vm, float Um, float Isyn, int m){
  return (ks[m] * (Vm - Vrs[m]) * (Vm - Vts[m]) - Um + Ies[m] + Isyn) *
time_step / Cms[m];
}

float izhik_Um(float Vm, float Um, int m){
  return as[m] * (bs[m] * (Vm — Vrs[m]) — Um) * time_step;
}
```

```
for (int t = 0; t < Tsim; t++){

for (int n = 0; n < Nneur; n++){
  v1 = izhik_Vm(Vm, Um, Isyns[n], n);
  u1 = izhik_Um(Vm, Um, n);
  Vms[n] = Vm + v1*0.5f;
  Ums[n] = Um + u1*0.5f;

  v2 = izhik_Vm(Vms[n], Ums[n], (Isyn_new + Isyns[n])*0.5f, n);
  u2 = izhik_Um(Vms[n], Ums[n], n);
  Vms[n] = Vm + v2*0.5f;
  Ums[n] = Um + u2*0.5f;

  v3 = izhik_Vm(Vms[n], Ums[n], (Isyn_new + Isyns[n])*0.5f, n);
  u3 = izhik_Um(Vms[n], Ums[n], n);
  Vms[n] = Vm + v3;
  Ums[n] = Um + u3;

  v4 = izhik_Vm(Vms[n], Ums[n], Isyns[n], n);
  u4 = izhik_Um(Vms[n], Ums[n], n);
  Vms[n] = Vm + (v1 + 2.0f*(v2 + v3) + v4)*0.16666666f;
  Ums[n] = Um + (u1 + 2.0f*(u2 + u3) + u4)*0.16666666f;

  Isyns[n] = -(AMPA_Amuont[n])*(Vms[n] - Erev_exc[n])
           - GABA_Amuont[n]*(Vms[n] - Erev_inh[n]);

  if (Vm > Vpeaks[n]){
     spk_times[Nneur*neur_num_spks[n] + n] = t;
     neur_num_spks[n]++;
     Vms[n] = cs[n];
     Ums[n] = Um + ds[n];
  }

}


for (int c = 0; c < Ncon; c++){
  xs[c] = (xs[c] - weights[c])*exp_recs[c] + weights[c];
  us[c] = us[c]*exp_facs[c];
  if (spk_times[Nneur*syn_num_spks[c] + pre_syns[c]] == t - delays[c]){
     us[c] += Us[c]*(1.0f - us[c]);
     float delta_x = xs[c]*us[c];
     xs[c] -= delta_x;
     syn_num_spks[c]++;
     if (receptor_type[c] == AMPA_RECEPTOR){
          AMPA_Amuont[post_syns[c]] += delta_x;
     } else if (receptor_type[c] == GABA_RECEPTOR){
          GABA_Amuont[post_syns[c]] += delta_x;
     }
  }
}}
```

На каждом шаге интегрирования сначала производиться проход по всем нейронам с расчётом динамики и проверкой на генерацию спайка, в случае если

зафиксирована генерация спайка время генерации записывается в массив ***spike_times*** по индексу зависящему от порядкового номера спайка сгенерированного этим нейроном и индекса самого нейрона. Этот массив в дальнейшем используется для расчёта динамики синапсов.

Для синапсов же на каждом временном шаге проверяется количество спайков которое уже он обработал и если оно меньше чем количество спайков сгенерированных пресинапсом, то производиться проверка задержки между нейронами и если разница между текущим временем и временем генерации спайка на пресинапсе равна задержке то производиться инкрементация синаптических переменных ***u*** и ***x*** согласно формуле (4). А далее инкрементируется переменная задающая количество нейромедиаторов в синаптической щели ***AMPA_Amount*** или ***GABA_Amount*** в зависимости от того какой это тип синапса тормозный или возбуждающий. Количество нейромедиаторов пропорционально синаптическому току.

Поскольку вычисления динамики для каждого отдельного нейрона на каждом временном шаге происходят независимо друг от друга (каждый из нейронов обращается для чтения и записи в свои ячейки памяти, которые не пересекются с другими нейронами) то расчёт их динамики можно легко распараллелить с использованием технологии GPGPU. Расчёт динамики синапсов происходит похожим образом, единственна операция где разные синапсы могу обратиться к одной и той же ячейке памяти это приращение постсинаптического тока, поэтому при распараллеливании данная операция должна производиться атомарно.

Общие сведение о технологии GPGPU и CUDA

Технология GPGPU

Использование графических ускорителей для выполнения задач общего назначения в частности для математических расчётов или по английски GPGPU (General-purpose computing on graphics processing units) стало развиваться совсем недавно вместе со стремительным ростом вычислительной мощности видеопроцессоров. Большая вычислительная мощность связана с рядом особенностей видеопроцессоров по сравнению с центральными процессорами, в отличии от последних графические процессоры изначально создавались как суперпараллельные архитектуры для обработки графики. Для этих задач нужно было выполнять множество операций связанных с преобразованием матриц и такие операции легко поддавались параллелизации. Доведя процесс управления созданием и исполнением потоков на аппаратный уровень, графические процессоры позволили значительно оптимизировать задачи по обработке графической информации, практически сняв эту нужду с плечь центрального процессора. Однако со временем стало понятно, что ряд задач в приложениях имеют ту же специфику и могут быть гораздо быстрее решены с использованием графических процессоров (далее ГПУ). Список задач для ускорения которых в настоящее время применяются GPGPU вычисления:

- Задачи молекулярной динамики
- Астрофизика (гравитационная задача N-тел)
- Обработка изображений
- Электростатические задачи связанные с вычислением электрического поля по закону Кулона
- Моделирование механики твёрдого тела
- Моделирование спайковых нейронных сетей

Нас больше всех интересуют задача моделиspования спайковых нейронных сетей, применение технологии для решения этой задачи GPGPU позволяет

значительно сократить время вычисления по ряду причин, о которых будет сказано ниже.

Первоначально программы производящие GPGPU вычисления писались на шейдерных языках. Это накладывало ряд ограничений и доставляло некоторые неудобства. Например даже для выполнения простых операций вроде сложения 2-х матриц приходилось передавать данные в шейдерную программу в виде текстур, далее после выполнения операции сложения результат сохранялся выходной буфер, что требовало отрисовки фигуры на экране. Поэтому с целью обойти эти ограничения и позволить использовать весь потенциал видеоускорителей были разработаны различные аппаратно программные архитектуры позволявшие абстрагируясь от графической обработки и от внутреннего устройства видеоустройства сосредоточится непосредственно на самих вычислениях. Наиболее известные из таких технологий: CUDA от компании Nvidia, AMD Brook+ (уже не поддерживается), OpenCl (платформеннонезависимое расширение языка Си который может работать как с GPU так и с CPU).

В данном учебно-методическом пособии мы будем рассматривать примеры на основе технологии CUDA, поэтому далее будем рассматривать только её. Главное отличие программирования на языке CUDA Си для GPU от программирования с CPU исходит из того, что графический процессор представляет из себя суперпараллельное вычислительное устройство состоящее из огромного количества вычислительных ядер. Поэтому для выполенения операций в нём используется SIMT принцип (Single instruction, multiple threads одна инструкция , множество потоков). На каждом вычислительном блоке запускается отдельные поток которые выполняют похожие операции, но с данными из различных областей памяти. Такой подход является обычным для графических алгоритмов, но для научных задач требует своеобразного подхода к написанию вычисляющей программы. Зато позволяет значительно увеличить

количество исполняющих вычислительных блоков расположенных на одном устройстве за счёт их однотипности.

Перечислим самые главные различия архитектур CPU и GPU. Ядра центрального процессора созданы для исполнения с максимальной скоростью одного потока последовательных команд, а видеопроцессоры рассчитаны для быстрого исполнения с максимальной производительностью одновременно исполняемых инструкций. Разработчики центальных процессоров тоже пытаются достичь исполнения как можно большего числа инструкций параллельно, например существуют набор инструкций SSE2, SSE3 и т. д. позволяющих исполнять несколько команд за такт, но поскольку поток данных остаётся последовательным, возможности этих расширений по ускорению вычислений ограничены.

У GPU же изначально параллельная архитектура. Графический ускоритель принимает на вход группу полигонов в виде массива данных, и после проведения необходимых операций выдаёт пиксели. Обработка каждого пикселя производится параллельно, независимо друг от друга. Поэтому по причине изначально параллельной организации выполнения инструкций на GPU используется большее количество вычислительных блоков. Более того, некоторые видеопроцессоры могут выполнять более одной инструкции за такт.

Ещё одним отличием графических процессоров от центральных является принцип доступа к памяти. В GPU он гораздо более предсказуем и максимально быстр в случае если несколько потоков одновременно обращаются к соседним элементам глобальной памяти. Также в графических ускорителях обычно установлена более быстродействующая память, поэтому видеопроцессору доступна более быстродействующее хранилище данных, что весьма актуально при высокопроизводительных вычислениях с огромными потоками данных.

Как уже было сказано в видеопроцессорах относительное количество транзисторов занятых вычислением больше чем в процессорах общего назначения, это связано с тем что в последних большое количество транзисторов идут на буферы команд, на большое количество кэш памяти,

расположенной прямо на чипе и на блоки предсказания ветвления. В видеопроцессорах же значительная часть площади чипа тратиться на вычислительные ядра. Планировщики исполнения потоков и разделямая память (*shared memory* аналог кэша в GPU) занимают незначительный объём. Данные особенности не позволяют ускорить выполнения каждого потока, но с их помощью можно запускать огромное количество потоков исполняющихся параллельно и переключение между исполнениями которых осуществляться на уровне железа. После того как один процесс запрашивает данные из памяти вместо холостого ожидания планировщик запускает исполнение на время другого потока, этот процесс не является настолько трудоёмким как в случае между переключениями между исполнениями различных потоков на одной центральном процессоре. CPU исполняет 1-2 потока на ядро, а GPU могут исполнять до 1024 и более потоков на каждый мультипроцессор, количество которых в видеороцссоре может достигать десятков. А переключения между потоками осуществляет за несколько тактов, в то время как на CPU переключение занимает нескольких сотен тактов.

Другое отличие GPU от CPU заключается в том, что в последних используются SIMD (одна инструкция выполняется над множеством данных) блоки для векторных вычислений, в то время как в видеопроцессорах применяются SIMT (одна инструкция на множество потоков) блоки для выполнения потоков, в этом случае не требуется преобразование данных в векторы, поэтому допустимы произвольные ветвления внутри потоков.

Вкратце можно резюмировать, что GPU предназначены для параллельных вычислений с большим количеством операций, и значительно большая доля транзисторов в нём работают непосредственно по назначению — произведение операций по обработке данных, а не над управлением исполнения последовательных вычислений как в CPU.

Рисунок 11 — сравнение места занимаемого вычислительными устройствами и дополнительной логиков в CPU и GPU

Особенно заметен прирос производительности в случае расчётов на GPU задач требующих выполнения трудоёмких, с вычислительной стороны операций, при редких обращениях к памяти. Иначе говоря отношения количества тактов для выполнения операции к количеству обращений в память велико.

По указанным выше причинам в ряде задач вычисления на GPU совершаются гораздо быстрее чем с использованием центральных процессоров. Компания Nvidia приводит следующий график увеличения теоретической производительности в Гфлопс/с (количество операций с плавающей точкой) для различных процессоров общего назначения от компании Intel и для видеопроцессоров производства Nvidia с поддержкой CUDA по годам.

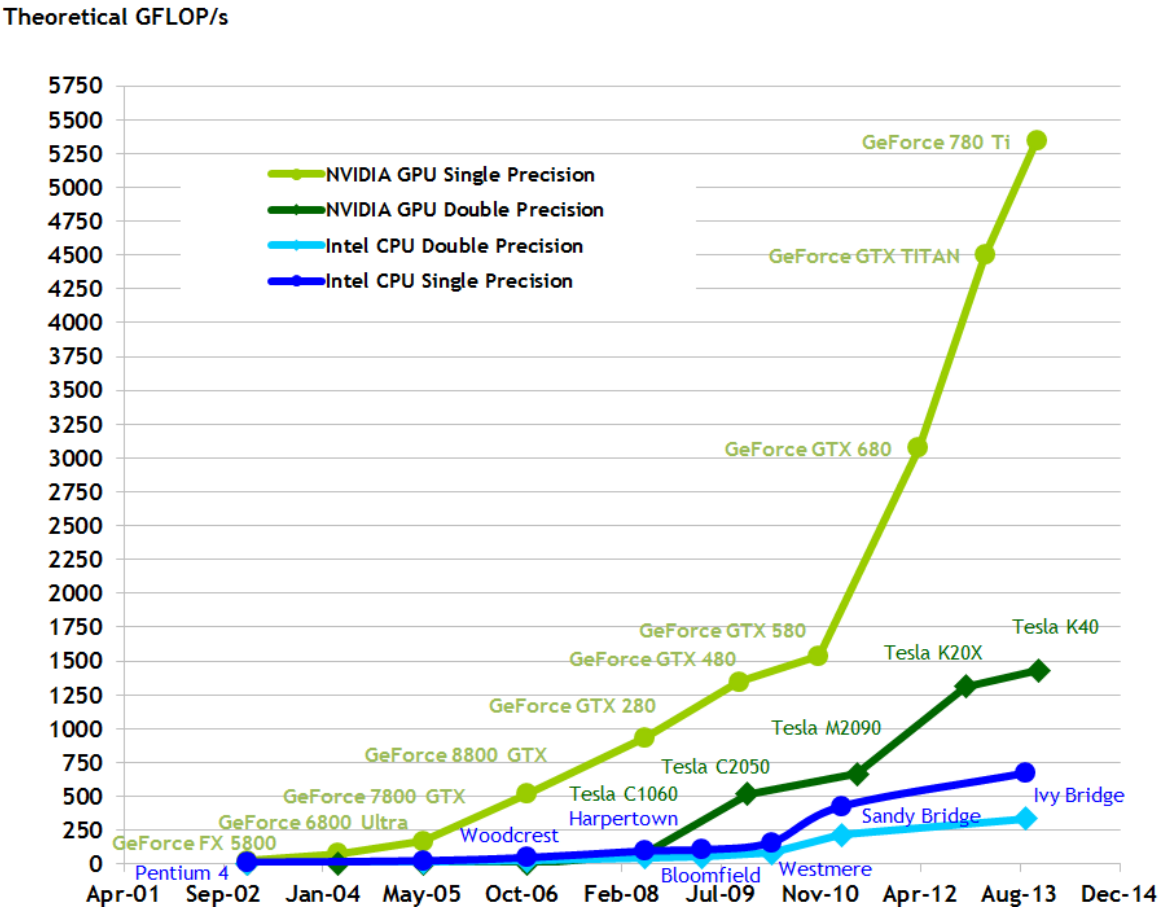

Рисунок 12 — Изменение производительности GPU производства Nvidia, и для процессоров компании Intel. (*Данные с сайта компании Nvidia*)

Программно-аппаратная архитектура CUDA

Технология CUDA (Cornpute Unified DeviceArchitecture) — это программно-аппаратная вычислительная архитектура, включающая в себя расширение языка Си, называемое CUDA Си, который включает в себя возможность организации доступа к вычислительным возможностям графического ускорителя и функции для работы с его памятью с целью выполнения параллельных высокопроизводительных вычислений. Технология CUDA присутствует во всех современных видеоускорителях компании Nvidia, начиная с 8-го поколения поколения, GeForce 8xxx.

Хотя программирование на GPU при помощи CUDA является более трудоёмким чем обычное программирование последовательных приложений, оно гораздо проще чем с ранними GPGPU решениями на базе шейдеров. Программирование на CUDA Си как и любое параллельное программирование требует разбиения задачи на несколько подзадач каждая из которых будет исполнятся независимо, однако в отличии от других технологий параллельного программирования вроде OpenMP и MPI существует ряд особенностей.

Инструментарий разработки состоит из языка CUDA Си, из Toolkt-а включающего в себя компилятор **nvcc** основанный на Open64 и различные инструменты по отладке и профилировке приложений. На сайте производителя содержится множество документации и примеров хорошо документированного кода.

Основные характеристики CUDA:

•унифицированное программно-аппаратное решение для параллельных вычислений для устройств NVIDIA;

•расширенный стандартный язык программирования Си (CUDA Си);

•стандартные библиотеки численного анализа FFT (быстрое преобразование Фурье) и BLAS (линейная алгебра);

•поддержка 32- и 64-битных операционных систем: Windows XP, Windows 7, Windows 8, Linux и MacOS X;

•возможность разработки на низком уровне.

Среда разработки CUDA (CUDA Toolkit) включает:

•компилятор nvcc;

•библиотеки FFT и BLAS;

•профилировщик;

•отладчик gdb для GPU;

•CUDA runtime драйвер в комплекте стандартных драйверов NVIDIA

•руководство по программированию;

•CUDA Developer Kit (исходный код, утилиты и документация).

CUDA включает два API: высокого уровня (CUDA Runtime API) и низкого (CUDA Driver API), в одной программе одновременное использование обоих невозможно. Высокоуровневый API работает «сверху» низкоуровневого, runtime вызовы транслируются в простые инструкции, обрабатываемые низкоуровневым Driver API. Но даже при использовании API высокоуровнего API необходимы знания о внутреннем устройстве и работе видеочипов NVIDIA.

GPU состоит из нескольких (от 2-х до 64-х) потоковых мультипроцессоров (Streaming Multiprocessor), в каждом из которых расположено от 32-128 до 1024 CUDA ядер (CUDA cores). Все инструкции выполняются по принципу одна инструкция применяется ко всем потокам в warpe, это группа из 32 потоков выполняемых физически одновременно. Поскольку не каждый поток внутри блока выполняется одновременно, такой способ выполнения называется SIMT (single instruction multiple threads — одна инструкция и много потоков).

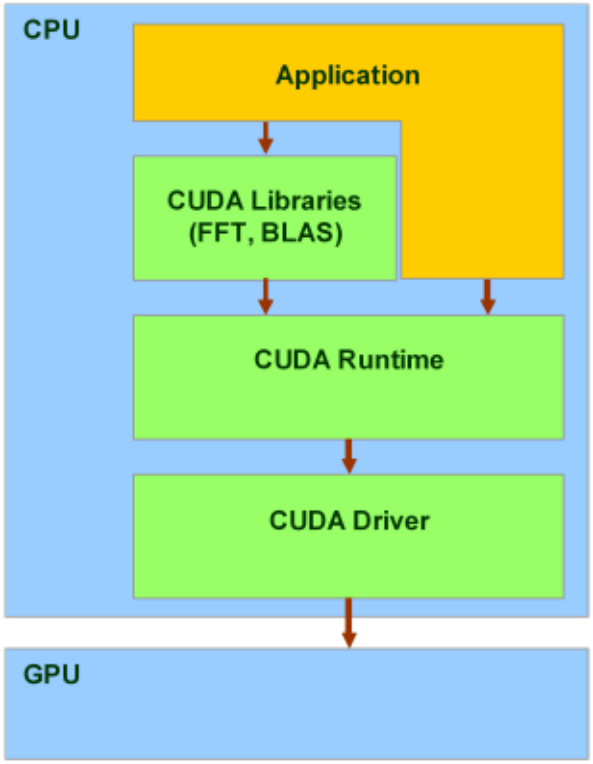

Рисунок 13 — Высокоуровневой CUDA Runtime API и низкоуровнево CUDA Driver API.

Каждый из мультипроцессоров имеет определённые ресурсы. Так, есть специальная разделяемая память объемом 16 килобайт на мультипроцессор, в современных архитектурах может доходить до 64 килобайт. Эта разделяемая память позволяет обмениваться информацией между потоками одного блока. Важно, что все потоки одного блока всегда выполняются одним и тем же мультипроцессором. А потоки из разных блоков обмениваться данными не могут, и нужно помнить это ограничение. Разделяемая память часто бывает полезной. Мультипроцессоры могут обращаться и к видеопамяти, но с большими задержками и худшей пропускной способностью. Для ускорения доступа и снижения частоты обращения к видеопамяти, у мультипроцессоров есть L1 кэша размером в 16 Кб/48 Кб в зависимости от настроек (физическая она расположена в одной схеме с разделяемой памятью и соотношение между ними программно можно регулировать) и версии архитектуры.

Максимальное число потоков запускаемых на одном видеопроцессоре

может достигать десятков тысяч. Для современного видеоадаптера серии GeForce GTX 760 количество физических cuda ядер составляет 1152 на 6 мультипроцессоров, при этом на каждом мультипроцессоре может быть запущено не более 2048 потоков. Знание этих ограничений необходимо для оптимизации алгоритмы под доступные ресурсы.

Модель программирования CUDA

Как уже упоминалось CUDA использует параллельную модель вычислений, каждый процессор выполняет одну и ту же операцию над разными данными в памяти параллельно. Графический процессор выступает в роли сопроцессора (device) по отношению к центральному процессору (host), который обладает большей памятью и обрабатывает параллельно большое количество потоков. Функция которая исполняется в каждом потоке называется ядром (kernel), она запускается с центрального процессора (host), однако в новых версиях начиная с coumpute capability 3.5 возможен и запуск kernel-функций изнутри другой kernel-функции.

Ранее уже упоминалось, что видеопроцессор отличается от центрального процессора тем, что может вычислять одновременно десятки тысяч потоков, что является обычным дело в графических приложениях, которые хорошо распараллеливается. Количество логических потоков и блоков потоков может превосходить количество физических исполнительных устройств, благодаря чему обеспечивается достаточно хорошая масштабируемость.

Согласно архитектуре CUDA потоки объединяются в блоки (thread block) — которые представляют из себя одномерные, двумерные или трёхмерные сетки потоков, которые могут взаимодействовать между собой при помощи разделяемой памяти и синхронизации (функция __syncthreads()). Функция kernel исполняется на сетке блоков потоков, рисунок 14. Каждый блок может быть 1D, 2D или 3D мерным, и может включать в себя 512 или 1024 потока в зависимости от версии архитектуры.

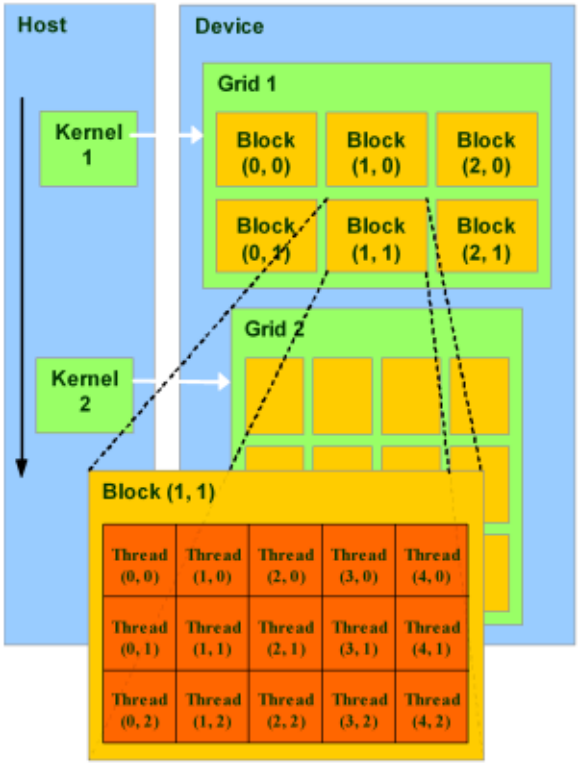

Рисунок 14 — Архитектура вычислений в CUDA. Вычислительные потоки и блоки.

Модель памяти CUDA

Каждый вычислительный поток имеет доступ к следующим типам памяти:

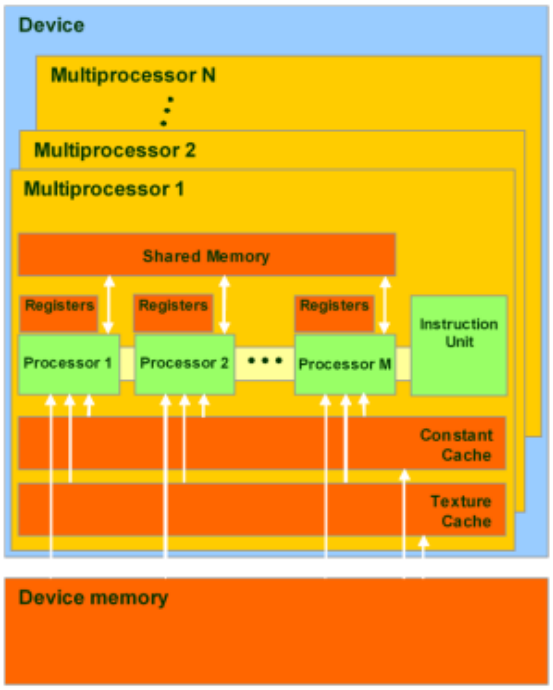

Рисунок 15 — Схематичное изображение программного представления видеопроцессора с указанием различных типов памяти.

Глобальная память — самый большой объём памяти, доступный для всех мультипроцессоров на видеочипе, размер составляет от 256 Мгб до 2 Гб на текущих решениях Ge Force GTX 760. Обладает высокой пропускной способностью, более 192 Гб/с для топовых решений NVIDIA, но очень большими задержками в несколько сот тактов.

Локальная память — это небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная — такая же, как и глобальная, но кэшурется.

Разделяемая память — это блок памяти объёмом 16 Кб / 48 Кб с общим доступом для всех потоков внутри блока. Эта память весьма быстрая, такая же, как регистры. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую и имеет низкие задержки. Преимущества разделяемой памяти: использование в виде управляемого программистом кэша первого

уровня, снижение задержек при доступе исполнительных блоков (ALU) к данным, сокращение количества обращений к глобальной памяти.

Память констант — область памяти объемом 64 килобайта, доступная только для чтения всеми мультипроцессорами. Она кэшируется по 8 килобайт на каждый мультипроцессор. Довольно медленная — задержка в несколько сот тактов при отсутствии нужных данных в кэше.

Текстурная память — блок памяти, доступный для чтения всеми мультипроцессорами. Выборка данных осуществляется при помощи текстурных блоков видеочипа, поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кэшируется по 8 килобайт на каждый мультипроцессор. Медленная, как глобальная — сотни тактов задержки при отсутствии данных в кэше.

Глобальная, локальная, текстурная и память констант — это физически одна и та же память, известная как локальная видеопамять видеокарты. Их отличия в различных алгоритмах кэширования и моделях доступа. Центральный процессор может обновлять и запрашивать только внешнюю память: глобальную, константную и текстурную.

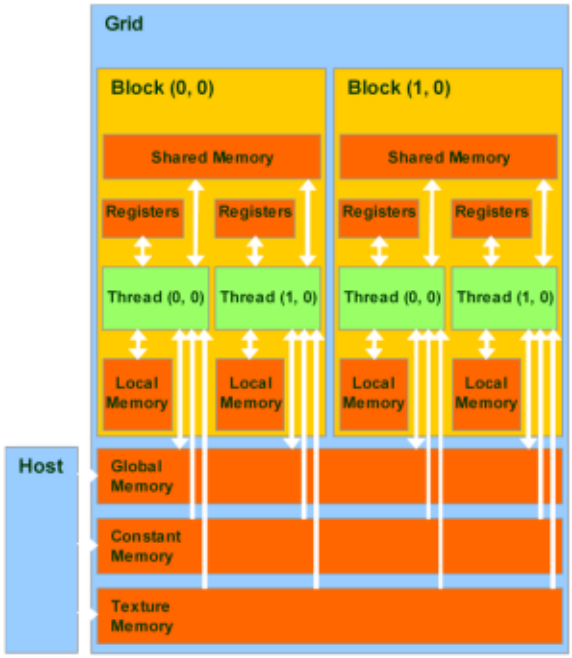

Рисунок 16 — Схематическое изображение программных блоков и потоков и их

взаимодействия с памятью видеопроцессора.

Из написанного выше понятно, что CUDA предполагает специальный подход к разработке, не совсем такой, как принят в программах для CPU. Нужно помнить о разных типах памяти, о том, что локальная и глобальная память не кэшируется и задержки при доступе к ней гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах.

Типичный, но не обязательный шаблон решения задач:

- задача разбивается на подзадачи;
- входные данные делятся на блоки, которые вмещаются в разделяемую память;
- каждый блок обрабатывается блоком потоков;
- подблок подгружается в разделяемую память из глобальной;
- над данными в разделяемой памяти проводятся соответствующие вычисления;
- результаты копируются из разделяемой памяти обратно в глобальную.

Ряд задач могут быть решены и без привлечения разделяемой памяти. В частности в случае моделирование спайковых нейронных сетей в которых влияние одних нейронов на другие опосредуется посредством времён спайков и задержек их распространения использование разделяемой памяти едва ли возможно.

Написание программ на CUDA и распараллеливание расчёта динамики спайковой сети

Программная часть технологии CUDA для разработчиков включает в себя: компилятор nvcc который компилирует программы написанные на языке Си со специальными расширениями который обычно называет CUDA Си, CUDA DriveAPI и CUDA Runtime API. В данном пособии мы разрберём только написание приложений с использованием Runtime API.

Специальные расширения CUDA Си включают в себя: встроенные переменные для идентификации номера потока, ряд функций для обращения с памятью видеоустройства и запуска вычислений на нём и спецификаторы функций и переменных. По терминологии CUDA центральный процессор называется host'ом, а видеопроцессор device'ом Существуют 3 типа спецификаторов функций:

`__global__` – вызываются с хоста выполняются на GPU

`__host__` – вызываются и выполняются на хосте, по умолчанию любые объявленные функции имеют такой спецификатор

`__device__` – вызываются и выполнят только на GPU.

Возможна комбинация спецификаторов, **`__host__`** и **`__device__`** в этом случае будет скомпилировано 2 варианта функции, одна для GPU и другая для CPU.

Для запуска вычислений на видеопроцессоре необходимо написать функцию кернел, данная функция должна имеет спецификатор **`__global__`** возвращять **void**. При запуска функции такого рода нужно указать количество потоков в блоке и количество блоков, при её исполнении видеопроцессор сам распределит нагрузку на все вычислительные ядра. Внутри такой функции можно обращаться к специальным переменным **`blockIdx`**, **`blockDim`**, **`threadIdx`** в которых храниться текущий номер блока и номер потока. Код программы должен иметь расширение *.cu.

Спецификаторы переменных:

__device__ – объявление переменной с этим идентификатором обозначает, что она будет храниться в памяти видеопроцессора, и доступна всем потокам выполняемым на нём, доступ к такой переменной с хоста возможен только с применением функций типа **cudaMalloc**, **cudaMemcpy**.

__shared__ – указывает на то, что переменная будет храниться на GPU в области быстрой разделяемой памяти, но доступ к ним возможен только внутри одного блока, в данном симуляторе такая память не используется.

__constant__ – указывает на то, что переменная храниться в константной памяти которая также является достаточно быстрой по сравнению с **device** памятью, однако в данной программе она тоже не использовалась.

Для резюмирования вышеуказанных особенностей cuda ниже приведён листинг программы осуществляющей сложение 2-х матриц с использованием вычислительных возможностей GPU.

```
/* sumArr.cu */
#include <stdio.h>

#define BLOCK_SIZE  32
#define N           64

__global__ void matMult(float * a, float * b, int n, float * c){
  // block index
  int   bidx  = blockIdx.x;
  // thread index
  int   tidx  = threadIdx.x;
  int   idx  = BLOCK_SIZE * bidx + tidx;

  c[idx] = a[idx] + b[idx];
}

int main(int argc, char *  argv []){
  size_t numBytes = N * sizeof(float);

  float* adev = NULL;
  float* bdev = NULL;
  float* cdev = NULL;
  cudaMalloc((void**) &adev, numBytes);
  cudaMalloc((void**) &bdev, numBytes);
  cudaMalloc((void**) &cdev, numBytes);

  float* a = new float[N];
  float* b = new float[N];
```

```
  for ( int i = 0; i < N; i++ ){
    a[i] = 1.0f;
    b[i] = 0.0f;
  }

  cudaMemcpy(adev, a, numBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(bdev, b, numBytes, cudaMemcpyHostToDevice);

 // запуск вычислительного ядра
  matMult<<<N/BLOCK_SIZE, BLOCK_SIZE>>> (adev, bdev, N, cdev);

  // Получение результатов из глобальной памяти
  float * c = new float[N];
  cudaMemcpy(c, cdev, numBytes, cudaMemcpyDeviceToHost);

  for (int i = 0; i < N; i++){
    printf("a:%.2f b:%.2f c:%.2f \n", a[i], b[i], c[i]);
  }

  cudaFree(adev);
  cudaFree(bdev);
  cudaFree(cdev);
  delete a;
  delete b;
  delete c;

  return 0;
}
```

Используя данный подход можно модифицировать алгоритм расчёта динамики спайковой нейронной сети представленный в прошлый главах. Как уже было упомянуто, для распараллеливания вычислений вместо прохождения по циклам можно запустить расчёт динамики каждого отдельного нейрона и синапса отдельным потоком, каждый из которых будет производить вычисления независимо друг от друга не производя одновременно операций чтения/записи в одни и те же адреса памяти. В нашем представлении индекс нейрона/синапса будет равен индексу потока. Ввиду того, что необходимо сохранять значения мембранных потенциалов между запусками ядра они будут размещены в глобальной памяти. Таким образом, часть глобальной памяти где будут храниться нейроны будет выглядеть следующим образом.

Thread index	0	1	2	...	N
	Vm[0]	Vm[1]	Vm[2]	Vm[...]	Vm[N]
	Um[0]	Um[1]	Um[2]	Um[...]	Um[N]
	...				

Рисунок 17 — размещение переменных нейрона в глобальной памяти GPU

Единственное критичное место где параллельность прерывается, это учёт влияния спайков одного нейрона на другой, осуществляемый в потоках синапсов. Если в один и тот же момент времени интегрирования на один и тот же постсинаптический нейрон приходят спайки с разных нейронов, приращение количества нейротрансмиттера **GABA_Amuont/AMPA_Amuont** должно осуществляется атомарно (это обозначает, что пока один поток не закончил операцию над этой ячейкой памяти другие потоки также пытающиеся совершить операцию над данной ячейкой памяти будут приостановлены).

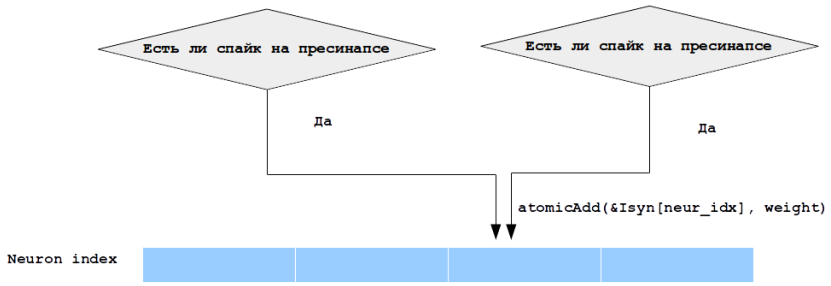

Рисунок 18 — Учёт влияния спайков с помощью атомарного инкрементирования

Кроме вышеуказанных особенностей остальные части алгоритма остаются прежними. Фрагмент программы рассчитывающей динамику спайковой нейронной сети параллельно с использованием GPU приведён ниже.

```
__global__ void integrate_neurons(float * Vms, float * Ums, ..){
  // определение номера потока который равен индексу нейрона
  int n = blockIdx.x*blockDim.x + threadIdx.x;
  v1 = izhik_Vm(Vm, Um, Isyns[n], n);
  u1 = izhik_Um(Vm, Um, n);
  Vms[n] = Vm + v1*0.5f;
```

```
  Ums[n] = Um + u1*0.5f;

  v2 = izhik_Vm(Vms[n], Ums[n], (Isyn_new + Isyns[n])*0.5f, n);
  u2 = izhik_Um(Vms[n], Ums[n], n);
  Vms[n] = Vm + v2*0.5f;
  Ums[n] = Um + u2*0.5f;

  v3 = izhik_Vm(Vms[n], Ums[n], (Isyn_new + Isyns[n])*0.5f, n);
  u3 = izhik_Um(Vms[n], Ums[n], n);
  Vms[n] = Vm + v3;
  Ums[n] = Um + u3;

  v4 = izhik_Vm(Vms[n], Ums[n], Isyns[n], n);
  u4 = izhik_Um(Vms[n], Ums[n], n);
  Vms[n] = Vm + (v1 + 2.0f*(v2 + v3) + v4)*0.16666666f;
  Ums[n] = Um + (u1 + 2.0f*(u2 + u3) + u4)*0.16666666f;

  Isyns[n] = -(AMPA_Amuont[n])*(Vms[n] - Erev_exc[n])
          - GABA_Amuont[n]*(Vms[n] - Erev_inh[n]);
  if (Vm > Vpeaks[n]){
     spk_times[Nneur*neur_num_spks[n] + n] = t;
     neur_num_spks[n]++;
     Vms[n] = cs[n];
     Ums[n] = Um + ds[n];
  }
}
__global__ void integrate_synapses(float * xs, float * Us, ..){
  // определение номера потока который равен индексу нейрона
  int c = blockIdx.x*blockDim.x + threadIdx.x;

  xs[c] = (xs[c] - weights[c])*exp_recs[c] + weights[c];
  us[c] = us[c]*exp_facs[c];
  if (spk_times[Nneur*syn_num_spks[c] + pre_syns[c]] == t -
  delays[c]){
     us[c] += Us[c]*(1.0f - us[c]);
     xs[c] -= xs[c]*us[c];

     syn_num_spks[c]++;
     if (receptor_type[c] == AMPA_RECEPTOR){
          atomicAdd(&AMPA_Amuont[post_syns[c]],  delta_x)
     } else if (receptor_type[c] == GABA_RECEPTOR){
          atomicAdd(&GABA_Amuont[post_syns[c]] ,  delta_x)
     }
  }
}
```

Запуск вычислительного ядра

```
integrate_neurons<<<Nneur/NEUR_BLOCK_SZ + 1, NEUR_BLOCK_SZ>>>(
          Vms_dev, Ums_dev, …, t)
    integrate_synapses<<<Ncon/SYN_BLOCK_SZ + 1,
SYN_BLOCK_SZ>>>(xs_dev, Us_dev, .., t)
```

Описание и архитектура программы-симулятора NNSim

Чтобы избежать каждый раз при моделирование написания громоздкого кода на CUDA Си была разработана программа-симулятор NNSim, поозволяющая производить конструирование и моделирование динамики спайковых нейронных сетей на языке Python. Работа с программой осуществляется с помощью скриптов, либо путём непосредственного вызова функций симулятора из интерактивной оболочки интерпретатора Python. Данная программа позволяет избегать написания излишнего кода выполняющего интегрирование уравнений спайковой сети, вместо этого можно сосредоточится непосредственно на исследовании их динамики и быстрой проверке гипотез возникающих в процессе исследования. Вычислительное ядро симулятора написано на CUDA Си и C++ а python используется для общения с конечным пользователем.

Поскольку написание дополнительных модулей расширяющих возможности симулятора с сохранением эффективности вычислений невозможно без знания внутренней архитектуры, оставшаяся глава будет посвящена именно этому.

Программа-симулятор состоит из следующих частей: модуль ***nnsim.py*** обеспечивающий программный интерфейс (API) для вызова функций симулятора пользователем из Python скриптов, модуль обёртка ***nnsim_pykernel.cpp*** используя который модуль ***nnsim.py*** вызывает C++ функции вычислительного ядра ***kernel_api.cpp***. Также в него входит ***cuda_kernel_api.cu*** который реализует вычисления на видеопроцессоре используя технологию CUDA от компании Nvdia. В случае отсутствия видеоускорителя на компьютере пользователя, можно проводить вычисления с использованием центрального процессора.

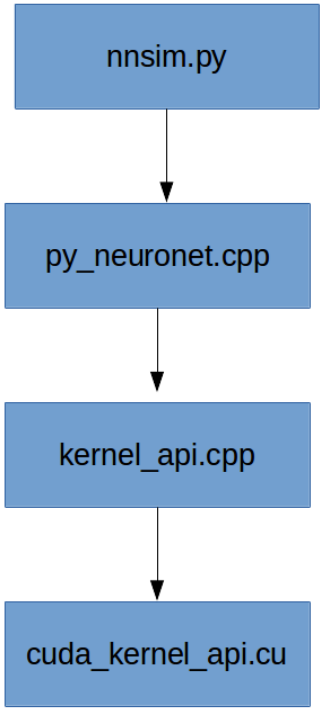

Рисунок 19 — Архитектура программы-симулятора NNSim.

В ходе моделирование динамики нейронов симулятор NNSim позволяет совершать запись осциллограмм отдельных нейронов/синапсов, а также осциллограмм отдельных групп нейронов/синапсов с одновременным усреднением на лету с целью уменьшения используемой памяти. Для примера если необходимо записать осциллограммы всех нейронов в сети а затем усреднить по всем нейронам с целью получения динамики средней активности сети, в случае если сеть состоит из 1000 нейронов и производится симуляция 1 с с шагом 0.1 мс в симуляторе nest потребуется ~100 мб, в симуляторе же NNSim в тысячу раз меньше, потому что усреднение по нейронам производится на каждом шаге интегрирования.

Примеры использования нейросимулятора NNSim

Рассмотрим несколько примеров использования нейросимулятора. Инструкция по установке и список всех функций приведены в приложении.

Пример 1. Моделирование низкоразмерной сети

Показать на примере нескольких нейронов явление кратковременной синаптической пластичности. Необходимо собрать сеть состоящую из 3-х нейронов, где 1 будет генерировать спайки с постоянной частотой, а 2 других будут синаптически связаны с ним, первый тормозной фасилитирующей связью, а второй возбуждающей депрессующей. Для выполнения этого задания создадим новый текстовой файл с расширением *.py и произвольным именем со следующим содержанием

```
# -*- coding: utf-8 -*-
"""
Example of using NNSim for modeling short term placticity
"""
# Импорт библиотеки симулятора
import nnsim

# Импорт различных библиотек
from nnsim import *
import matplotlib.pyplot as pl
import numpy as np

h          = 0.2      # шаг интегрирования
SimTime    = 2000.    # время симуляции в мс

# Нейрон генератор, от которого спайки будут идти к 2-м разным
нейронам
gnrtr = create(1, n_type="exc", Ie=35, d=100.)
# тормозный и возбуждающий нейрон, устновлены только времена
затухания
# постсинптических токов, остальные параметры по умолчанию
n_inh = create(1, n_type="inh", tau_psc_inh=15., tau_psc_exc=5.)
n_exc = create(1, n_type="exc", tau_psc_inh=15., tau_psc_exc=5.)

# соединение генератора с другим нейроном посредством тормозной
связи
con = connect(gnrtr, n_inh, weight=10., conn_spec='one_to_one',
syn='inh')
# соединение генератора с другим нейроном посредством возбуждающей
связи
con = connect(gnrtr, n_exc, weight=5., conn_spec='one_to_one',
```

```
syn='exc')
# задание списки нейронов с которых будет вестись запись
осциллограммы
neur_rec = n_exc + n_inh + gnrtr
record(neur_rec)

simulate(h, SimTime) # запуск симуляции без применения GPU

# получение осциллограмм и их рисование
(Vm, Um, Isyn, y_exc, y_inh, x, u) = get_results()
t = np.linspace(0, SimTime, len(Vm[0]))

pl.figure()
ax = []
for i in range(len(neur_rec)):
    ax.append(pl.subplot(len(neur_rec), 1, i+1))
    ax[i].plot(t, Vm[i])
    ax[i].set_ylabel("Vm_"+str(neur_rec[i]) + ", mV")
pl.xlabel("Time, ms")
pl.show()
```

далее выполним в терминале команду: ***python имя_файла.py*** и получим результат вычисления динамики мембранного потенциала 3-х тестовых нейронов, Рисунок 20

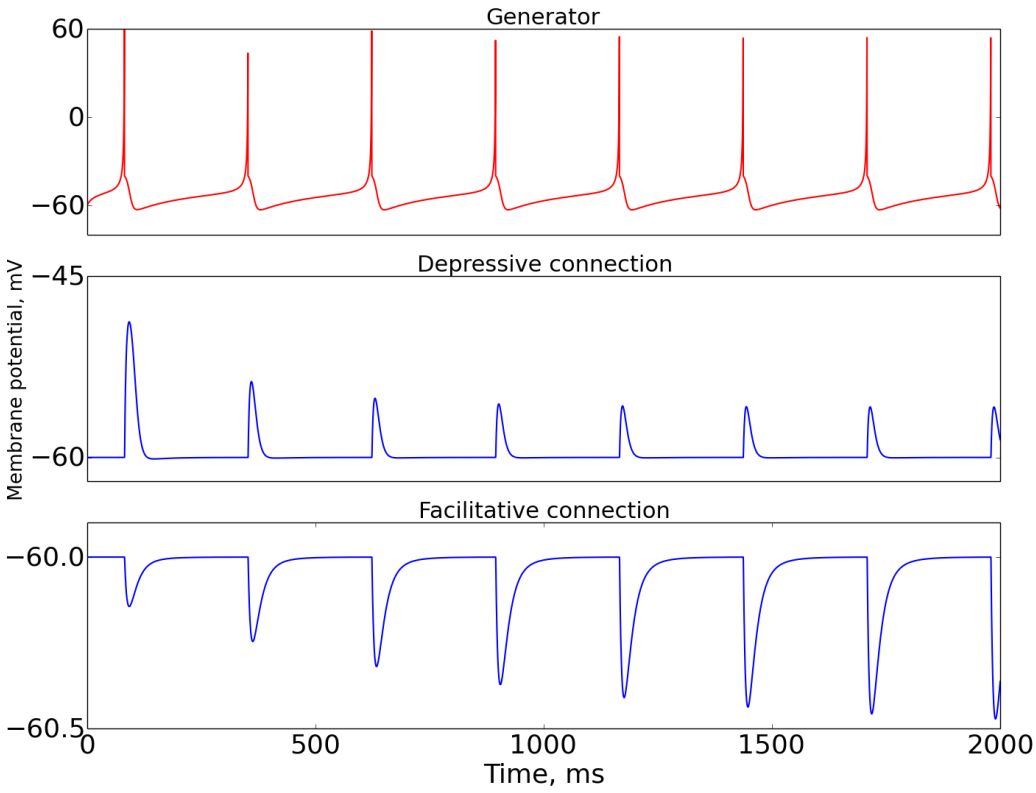

Рисунок 20 — динамика мембранного потенциала полученная в NNSim в случае наличия кратковременной пластичности.

Отчётливо видно, что с каждым новым спайком амплитуда ответа на постсинаптическом нейроне падает в случае депрессии и растёт в случае фасилитации. Параметры по умолчанию заданы такими, что тормозные связи являются фасилитирующими, а возбуждающие депрессующими. Но при использовании симулятора можно задать произвольные значения для времён восстановления нейромедиатора *tau_rec* и времени затухания *tau_fac* переменной *u* тем самым регулируя режим пластичности конкретной связи.

Пример 2. Моделирование крупномасштабной сети

Вычисление динамики крупномасштабной сети. Собрать спайковую сеть состоящую примерно из 2-х тысяч нейронов, продемонстрировать явление генерации популяционных разрядов. В этом случае ввиду наличия большого количества элементов существенное значение будет иметь время симуляции. В случае симуляции с применением технологий высокопроизводительных параллельных вычислений на видеопроцессоре оно должно происходить значительно быстрее. Скрипт на языке Python:

```
# -*- coding: utf-8 -*-
"""
Script for simulting large scale network with using GPU for
performing calculations
"""

from nnsim import *
import nnsim
import matplotlib.pyplot as pl
import matplotlib
import numpy as np
# установка толщины линии и размера шрифта по умолчанию
matplotlib.rc('lines', linewidth=1.5)
matplotlib.rc('font', size=26.)
h       = .5
SimTime = 4000.
bin_sz  = 5.0  # размер временног окна в котором будет считаться
кол-во спайков
NumExc  = 1400  # кол-во возбуждающих нейронов
NumPmkr = 100   # кол-во генераторов ритма (pacemaker)
NumInh  = 400   # кол-во тормозных нейронов
p_con   = 0.025 # вероятность 2-х нейронов быть связанными

# создние возбуждающей популяции, генераторов ритма (pmkr) и
тормозных нейронов
n_exc = create(NumExc, n_type="exc", d={'distr': 'normal', 'mean':
70., 'std': 25.})
n_pmkr = create(NumPmkr, n_type='exc', Ie={'distr': 'normal',
'mean': 35., 'std': 10.})
n_inh = create(NumInh, n_type="inh")

# расчёт среднего количества исходящих связей на каждый нейрон
# задннй популяции исходя из вероятности быть связанными
N_exc_con = int(round(np.sqrt(p_con) * NumExc + NumPmkr))
N_inh_con = int(round(np.sqrt(p_con) * NumInh))
```

```
# выбор из популяции возбуждающих нейронов N_exc_con случайных
нейронов
# каждый из которых будет связан с N_exc_con других нейронов из
всех нейронов
pre = np.random.permutation(n_exc + n_pmkr)[:N_exc_con]
con = connect(pre, n_inh+n_exc+n_pmkr, conn_spec={'rule':
'fixed_outdegree', 'N': N_exc_con},
              delay={'distr': 'uniform', 'low': 0., 'high': 10.},
              tau_rec=800.,
              x={'distr': 'uniform', 'low': 0., 'high': .5},
              weight={'distr': 'normal', 'mean': 7., 'std': 4.})
# тоже самое для тормозной популяции
pre = np.random.permutation(n_inh)[:N_inh_con]
con2 = connect(pre, n_inh+n_exc+n_pmkr, syn="inh",
conn_spec={'rule': 'fixed_outdegree', 'N': N_inh_con},
              delay={'distr': 'uniform', 'low': 0., 'high': 10.},
              x={'distr': 'uniform', 'low': 0., 'high': .5},
              weight={'distr': 'normal', 'mean': 6., 'std': 2.})
# замер времени симуляции
import time
start = time.time()
simulate(h, SimTime, gpu=False)
print "Elapsed %f s" % (time.time() - start)

# визуализация полученных данных
(times, senders) = get_ordered_spikes()
fig = pl.figure(figsize=(12, 9))
ax1 = pl.subplot(211)
ax2 = pl.subplot(212, sharex=ax1)
ax1.set_ylim([0, max(senders)])
ax1.plot(times, senders, '.k')
ax1.set_ylabel("Neuron #")
ax2.hist(times, range=(0, SimTime), bins=int(SimTime/bin_sz),
histtype='step')
ax2.set_ylabel("Num spikes in {0} ms".format(bin_sz))
ax2.set_xlabel("Time, ms")

pl.show()
```

Результаты счёта приведены на Рисунке 21. Отчётливо видно как высокосинхронные популяцинные разряды перемежаются с периодами относительно низкой активности сети. Такого рода активность является наиболее часто встречаемой в живых нейронных сетях. [4, 5]

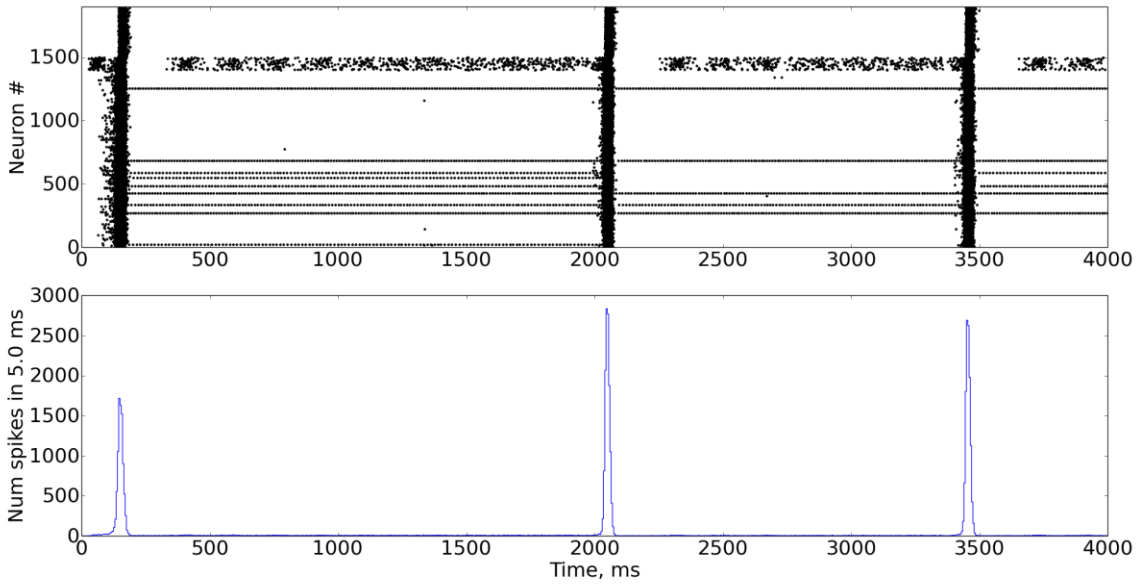

Рисунок 21 — Динамика крупномасштабной нейронной сети рассчитанная в NNSim. Вверху растр активности, по оси *x* время, по оси *y* номер нейрона, моменты возникновения спайков отмечены точкой. Внизу средняя сетевая активность, кол-во спайков во временном окне в 5 мс.

Сравнение скорости счёта для случаев запуска вычислений в системе с процессором Intel Core i7-2600, 3.4 GHz и видеопроцессором Nvidia GeForce 560 Ti для различного размера сети, рисунок 22. Параметры сети такие же как в предыдущей сети, вероятность связи постоянная, поэтому при увеличении числа нейронов количество связей растёт пропорциональную квадрату общего количества нейронов.

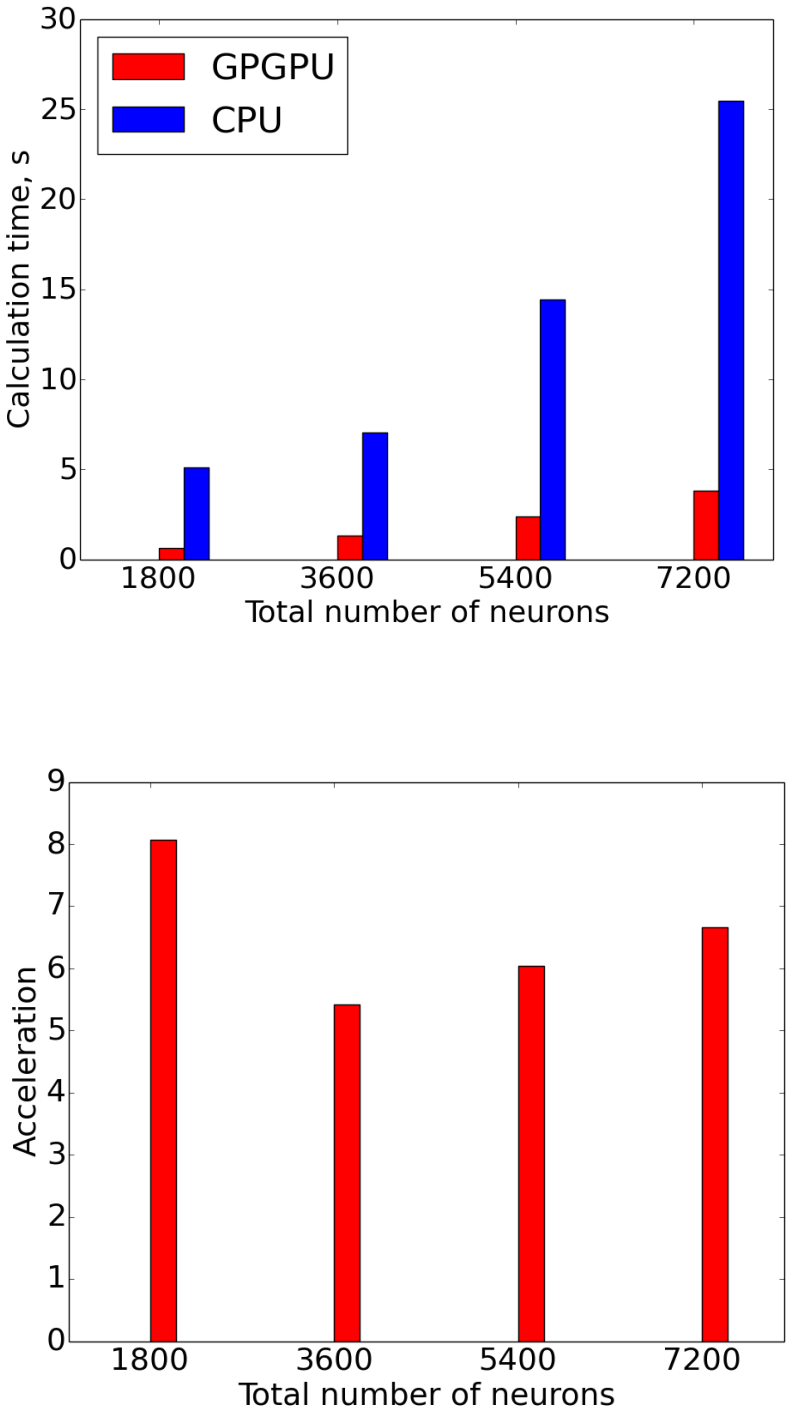

Рисунок 22 — Времена расчёта при вычислениях на центральном процессоре (CPU) и при использовании видеопроцессора (GPGPU) при различном количестве элементов в сети, справа достигнутый прирост производительности.

Приложения

Приложение 1: список функций симулятора NNSim с описанием

`create(N, n_type="exc", **kwargs)`

Создаёт популяцию из ***N*** нейронов. *****kwargs*** обозначает, что это функция с переменным количеством параметров. Можно указывать отдельные параметры нейронов, например: ***`create(10, Cm=55.)`***, в этом случае этот параметр будет одинаков для всех нейронов созданной популяции. Каждый отдельный параметр нейрона можно также задавать в виде случайного распределения, в этом случае аргументом должен быть словарь в котором будут заданы параметры распределения. Например: ***`create(10, Cm=55, d={'distr': 'normal', 'mean': 50., 'std': 5.})`***, где ключ ***`'distr'`*** задаёт тип распределения, его возможные значения: ***`'normal'`***, ***`'uniform'`***. Если распределение нормальное, то для него указываются среднее и стандартное отклонение, если равномерное то минимальное значение случайной величины ***`'low'`*** и максимальное ***`'high'`***. ***`n_type`*** задаёт тип нейрона в соответствии с которым будут установлены значения по умолчанию для параметров непосредственные значения которых не заданы при вызове функции. Возможные значения ***`'exc'`*** и ***`'inh'`*** соответствующие возбуждающим и тормозным нейронам. Функция возвращает список с индексами нейронов, которые в дальнейшем могут быть использованы при вызове **connect. record, mean_record.**

`connect(pre, post, conn_spec='one_to_one', syn='exc', **kwargs)`

Связывание нейронов с индексами из списка ***`pre`*** с нейронами из списка ***`post`***. Параметр ***`conn_spec`*** задаёт способ связывания. Если он равен ***`'one_to_one'`*** то каждый нейрон из ***`pre`*** соединяет с ***`post`*** в этом случае размерности ***`pre`*** и ***`post`*** должны быть равны, если

'all_to_all' то каждый нейрон из ***pre*** соединяется со всеми нейронами из ***post***. Также возможны сложные алгоритмы связывания, в этом случае ***conn_spec*** принимает в качестве значения словарь с параметрами соединений. Возможные значения ***conn_spec***:

{'rule': 'fixed_total_num', 'N': N} - будет создано ***N*** связей из ***N*** случайных комбинаций из ***pre*** и ***post***,

{'rule': 'fixed_outdegree', 'N': N} - каждый нейрон из ***pre*** будет иметь ***N*** исходящих связей с ***N*** случайными нейронами из ***post***,

{'rule': 'fixed_indegree', 'N': N} - каждый нейрон из ***post*** будет иметь ***N*** входящих связей из ***N*** случайных нейронов из ***post***,

{'rule': 'mean_outdegree', 'N_mean': N_mean, 'N_std': N_std} - количество исходящих связей каждого нейрона из ***pre*** будет нормально распределённым случайным числом с средним ***N_mean*** и стандартным отклонением ***N_std***,

{'rule': 'mean_indegree', 'N_mean': N_mean, 'N_std': N_std} - количество входящих связей на каждый нейрон из ***post*** будет нормально распределённым случайным числом с средним ***N_mean*** и стандартным отклонением ***N_std***. Также как и в случае создания нейронов можно задавать значения синаптических параметров в виде словаря если необходимо случайно распределить их, либо указать непосредственное значение если хотим чтобы параметр был одинаков для всех соединений. Параметр ***syn*** задаёт тип связи в соответсвии с которым устанавливаются значения по умолчанию для синаптических параметров непосредственные значения которых не заданы при вызове функции. Возможные значения *'exc'*, *'inh'* для возбуждающих и тормозных связей соответственно. Пример вызова функции:

```
connect(pre, post, conn_spec='one_to_one', syn='inh',
conn_spec={'rule': 'outdegree', 'N_mean': 100,
'N_std': 20}, x=0.5, u={'distr': 'uniform', 'low':
0., 'high': 1})-
```

соединит каждый нейрон из ***pre*** в среднем со 100 нейронами из ***post***, при этом на каждом нейроне это значение будет случайным и иметь стандартное отклонение 20, связи будут тормозными, переменная ***x*** для каждой связи будет иметь значение 0.5 а переменная ***u*** равномерно распределена от 0 до 1.

record(nodes, node_type='neur')

Добавляет элементы nodes в список элементов для которых будут сохранятся осциллограммы, в случае если ***node_type='neur'*** то ***nodes*** - это индексы нейронов, а в случае если ***node_type='syn'*** то это индексы синапсов. Осциллограммы в последствии могут быть возвращены с помощью функции. ***get_results***.

mean_record(nodes, node_type='neur', name=None)

Добавляет популяцию нейронов или синапсов (в случае если ***node_type='syn'***) в список для записи средних значений осциллограмм. Для каждой популяции можно указать имя, что позволит легче ориентироваться в полученных осциллограммах.

get_oscills(mean=False)

Возвращает массив ***(Vm, Um, Isyn, y_exc, y_inh, x, u)*** с осциллограммами мембранного потенциала и синаптических параметров для синапсов и нейронов которые были указаны до симуляции с помощью вызова функции ***record*** или ***mean_record***. Первый индекс в каждом таком массиве это индекс нейрона, а второй индекс это временной шаг. В случае, если ***mean=True*** возвращаются усреднённые по всей популяции осциллограммы, в этом случае первый индекс это порядковый номер популяции.

get_spk_times()

Возвращает список с временами спайков. Длинна списка равна количеству нейронов, при обращении к списку по индексу нейрон мы получаем список из времён спайков этого нейрона.

`get_ordered_spikes()`

Возвращает массив с временами спайков в формате 2-х списков, 1-й список это времена спайков, а 2-й список содержит индексы нейронов на которых произошли спайки. Максимальный индекс это число всех спайков во всей сети за время симуляции.

`simulate(h, SimTime, gpu=False)`

Запуск симуляции с временным шагом `h`, временем симуляции `SimTime` миллисекунд. Если параметр ***`gpu`*** равен ***`True`***, симуляция происходит с использование видеопроцессора, в противном случае вычисления производятся только на центральном процессоре. По умолчанию этот параметр равен ***`False`***.

Приложение 2: параметры по умолчанию для нейронов и синапсов

Параметры возбуждающих нейронов

```
neur_param['exc'] = {'a': 0.02, 'b_1': 0.5, 'b_2': 0.5, 'c': -40.,
'd': 100., 'k': 0.5, 'Cm': 50., 'Vr': -60., 'Vt': -45., 'Vpeak':
40., 'p_1': 1., 'p_2': 1., 'Vm': -60., 'Um': 0., 'Erev_AMPA': 0.,
'Erev_GABA': -70., 'Isyn': 0., 'tau_psc_exc': 3., 'tau_psc_inh':
7., 'Ie': 0., 'psn_seed': None, 'psn_rate': 0., 'psn_weight': 1.}
```

Параметры тормозных нейронов

```
neur_param['inh'] = {'a': 0.03, 'b_1': -2.0, 'b_2': -2.0, 'c': -
50., 'd': 100., 'k': 0.7, 'Cm': 100., 'Vr': -60., 'Vt': -40.,
'Vpeak': 35., 'p_1': 1., 'p_2': 1., 'Vm': -60., 'Um': 0.,
'Erev_AMPA': 0., 'Erev_GABA': -70., 'Isyn': 0., 'tau_psc_exc': 3.,
'tau_psc_inh': 7., 'Ie': 0., 'psn_seed': None, 'psn_rate': 0.,
'psn_weight': 1.}
```

Параметры возбуждающих синапсов

```
syn_param['exc'] = {'tau_rec': 800., 'tau_fac': 0.00001, 'U': 0.5,
'receptor_type': 1}
```

Параметры тормозных синапсов

```
syn_param['inh'] = {'tau_rec': 100., 'tau_fac': 1000., 'U': 0.04,
'receptor_type': 2}
```

Начальные условия для синаптических переменных

```
syn_default = {'y': 0., 'x': 1., 'u': 0., 'weight': 1., 'delay':
0.}
```

Приложение 3: установка симулятора

На операционных системах Windows 7 / 8

Программу-симулятор можно загрузить по ссылке github.com/esirpavel/nnsim/releases/. В ОС Windows сперва необходимо установить Python и Numpy python.org/downloads/windows/, sourceforge.net/projects/numpy/files/NumPy/1.9.1/. При установке следует выбрать пункт добавление python.exe в Path. Далее следует набрать в терминале:

```
pip install python-dateutil pyparsing six matplotlib
```

Эта команда установит библиотеку визуализации научных данных с матлабоподобным синтаксисом – Matplotlib.

В случае если пользователь не намеревается редактировать исходный код симулятора, можно просто скачать и установить исполняемые файлы NNSim-0.1.win32.exe или NNSim-0.1.win-amd64.exe. После этого можно пользоваться всеми возможностями симулятора после импорта библиотеки **`nnsim`** внутри python-скриптов или в интерактивной оболочке интерпретатора.

В случае если пользователь намерен редактировать исходный код программы ему нужно установить Cuda Toolkit с компилятором nvcc. Компилятор nvcc в Windows использует компилятор Майкрософ (msvc) для компиляции и сборки cuda проектов. Поэтому необходимо установить одну из версий компилятора Майкрософт. Наиболее удобный вариант установка компилятора вместе со средой Visual Studio Express. Данная урезанная версия среды разработки Visual Studio распространяется бесплатно. Версия Visual Studio Express Edition 2012 может быть загружена по адресу http://www.microsoft.com/ru-ru/download/details.aspx?id=34673. После установки необходимо зарегистрировать бесплатную копию среды разработки на сайте Майкрософт и ввести полученный на сайте ключ. При первом запуске программа сама направит по нужной ссылке.

После этого необходимо загрузить среду разработки Nvidia Cuda Toolkit по ссылке https://developer.nvidia.com/cuda-downloads.

Далее открыть командную строку cmd.exe и перейдя в директорию с распакованным симулятором nnsim ввести команду

```
nmake -f Makefile_win
```

Эта команда скомпилирует симулятор и создаст разделяемую библиотеку nnsim_pykernel.pyd, по сути это dll-файл с изменённым расширением для того, чтобы Python распознавал его. После этого запуская интерпретатор языка Python в одной папке с файлами nnsim_pykernel.pyd и nnsim.py можно пользоваться всеми функциями программы-симулятора. Для удобства можно скопировать эти файлы в папку включённую в PYTHONPATH. В Windows одна из таких папок *C:\Python27\Lib\site-packages*

На операционных системах семейства GNU/Linux

В GNU/Linux также существует возможность скачать установочный файл для соответствующего дистрибутива deb-based (Debian, Ubuntu, Linux Mint) или rpm-based (RedHat, Fedora, OpenSuse) по той же ссылке с github.com/esirpavel/nnsim/releases/. Также необходимо установить остальные пакеты **python, python-numpy, python-matplotlib** при помощи пакетного менеджера apt-get или yum. В deb-based дистрибутивах это делается командой:

```
sudo apt-get install python, python-numpy, python-matplotlib
```

В случае если пользователь желает редактировать исходный код симулятора, нужно с помощью пакетного менеджера установить также пакеты **python-dev, gcc, make.** Скачать CUDA Toolkit для Linux в виде deb пакета, потом обновить репозиторий командой **sudo apt-get update** и после этого набрать **sudo apt-get install cuda**. После этого симулятор можно компилировать командой **make**. Она создаст файл разделяемой библиотеки nnsim_pykernel.so. А вызов **sudo make install** скопирует их в папку с пакетами Python, чтобы можно было импортировать библиотеку **nnsim** из любого места.

Приложение 4: исходный код симулятора

```
/*
 * cuda_kernel_api.cpp
 *
 *  Created on: 01 нояб. 2014 г.
 *      Author: pavel
 */

#include "cuda_kernel_declarations.h"
#include "nnsim_constants.h"

__device__ float get_random(unsigned int *seed){
	// return random number homogeneously distributed in interval [0:1]
	unsigned long a = 16807;
	unsigned long m = 2147483647;
	unsigned long x = (unsigned long) *seed;
	x = (a * x) % m;
	*seed = (unsigned int) x;
	return ((float)x)/m;
}

__global__ void integrate_synapses(float* x, float* u, float* exp_rec, float*
exp_fac, float* U,
                                   float* weight, unsigned int*
delay, unsigned int* pre_syn, unsigned int* post_syn, unsigned int*
receptor_type,
                                   unsigned int* syn_num_spk,
unsigned int* neur_num_spk, unsigned int* spk_time,
                                   float* AMPA_Amuont, float*
GABA_Amuont,
                                   unsigned int t, int Ncon, int
Nneur){
	unsigned int c = blockDim.x*blockIdx.x + threadIdx.x;
	if (c < Ncon){
		x[c] = (x[c] - weight[c])*exp_rec[c] + weight[c];
		u[c] = u[c]*exp_fac[c];

		if (syn_num_spk[c] < neur_num_spk[pre_syn[c]]){
			if (t >= delay[c] && spk_time[Nneur*syn_num_spk[c] +
pre_syn[c]] == t - delay[c]){
//				printf("Spike! neur: %i time: %f\n", post_syns[c],
t*time_step);
				u[c] += U[c]*(1.0f - u[c]);
				float delta_x = x[c]*u[c];
				x[c] -= delta_x;
				syn_num_spk[c]++;

				// When run parallel this incrementation should be
atomic
				if (receptor_type[c] == AMPA_RECEPTOR){
					atomicAdd(&AMPA_Amuont[post_syn[c]], delta_x);
				} else if (receptor_type[c] == GABA_RECEPTOR){
					atomicAdd(&GABA_Amuont[post_syn[c]], delta_x);
				}
			}
		}

	}
}

__device__ __inline__ float check_pow(float x, float degr){
```

```
      if (degr == 1.0f){
            return x;
      } else {
            return powf(x, degr);
      }
}

__global__ void integrate_neurons(float* Vms, float* Ums,
            float* a, float* b1, float* b2, float* c, float* d, float* k, float*
p1, float* p2,
            float* Vpeak, float* Vr, float* Vt, float* Cm, float* Ie, float*
Isyn,
            float* AMPA_Amuont, float* GABA_Amuont, float* exp_pscs_exc, float*
exp_pscs_inh,
            float* Erev_exc, float* Erev_inh,
            float* y_psn, float* psn_weight, float* exp_psn, unsigned int*
psn_time, float* psn_rate, unsigned int* psn_seed,
            unsigned int* spk_time, unsigned int* neur_num_spk,
            unsigned int t, float time_step, int Nneur){
      unsigned int n = blockIdx.x*blockDim.x + threadIdx.x;
      float v1, u1, v2, u2, v3, u3, v4, u4;
      if (n < Nneur){
            y_psn[n] *= exp_psn[n];
            while (psn_time[n] == t){
                  y_psn[n] += psn_weight[n];
                  psn_time[n] -= -1 +
(1000.0f/(time_step*psn_rate[n]))*log(get_random(psn_seed + n));
            }

            AMPA_Amuont[n] *= exp_pscs_exc[n];
            GABA_Amuont[n] *= exp_pscs_inh[n];

            float Vm = Vms[n];
            float Um = Ums[n];
            // y_psns is here because poisson noise is excitatory
            float Isyn_new = -(AMPA_Amuont[n] + y_psn[n])*(Vm - Erev_exc[n]) -
GABA_Amuont[n]*(Vm - Erev_inh[n]);

            v1 = (k[n]*(Vm - Vr[n])*(Vm - Vt[n]) - Um + Ie[n] +
Isyn[n])*time_step/Cm[n];
            u1 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vm - Vr[n]),
p1[n]) - Um : b2[n]*check_pow((Vm - Vr[n]), p2[n]) - Um);
            Vms[n] = Vm + v1*0.5f;
            Ums[n] = Um + u1*0.5f;
            v2 = (k[n]*(Vms[n] - Vr[n])*(Vms[n] - Vt[n]) - Ums[n] + Ie[n] +
(Isyn_new + Isyn[n])*0.5f)*time_step/Cm[n];
            u2 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vms[n] -
Vr[n]), p1[n]) - Ums[n] : b2[n]*check_pow((Vms[n] - Vr[n]), p2[n]) - Ums[n]);
            Vms[n] = Vm + v2*0.5f;
            Ums[n] = Um + u2*0.5f;
            v3 = (k[n]*(Vms[n] - Vr[n])*(Vms[n] - Vt[n]) - Ums[n] + Ie[n] +
(Isyn_new + Isyn[n])*0.5f)*time_step/Cm[n];
            u3 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vms[n] -
Vr[n]), p1[n]) - Ums[n] : b2[n]*check_pow((Vms[n] - Vr[n]), p2[n]) - Ums[n]);
            Vms[n] = Vm + v3;
            Ums[n] = Um + u3;
            v4 = (k[n]*(Vms[n] - Vr[n])*(Vms[n] - Vt[n]) - Ums[n] + Ie[n] +
Isyn_new)*time_step/Cm[n];
            u4 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vms[n] -
Vr[n]), p1[n]) - Ums[n] : b2[n]*check_pow((Vms[n] - Vr[n]), p2[n]) - Ums[n]);
            Vms[n] = Vm + (v1 + 2.0f*(v2 + v3) + v4)*0.16666666f;
            Ums[n] = Um + (u1 + 2.0f*(u2 + u3) + u4)*0.16666666f;
```

```
            if (Vm > Vpeak[n]){
//                printf("Spike! neur: %i time: %f\n", n, t*time_step);
                  spk_time[Nneur*neur_num_spk[n] + n] = t;
                  neur_num_spk[n]++;
                  Vms[n] = c[n];
                  Ums[n] = Um + d[n];
            }
            Isyn[n] = Isyn_new;
      }
}

void simulateOnGpu(){
      init_mem();
      copy2device();
      for (unsigned int t = 0; t < Tsim; t++){
            integrate_neurons<<<Nneur/NEUR_BLOCK_SZ + 1, NEUR_BLOCK_SZ>>>(
                  Vms_dev, Ums_dev, as_dev, b1_s_dev, b2_s_dev, cs_dev, ds_dev,
ks_dev, p1_s_dev, p2_s_dev,
                  Vpeaks_dev, Vrs_dev, Vts_dev, Cms_dev, Ies_dev, Isyns_dev,
                  AMPA_Amuont_dev, GABA_Amuont_dev, exp_pscs_exc_dev,
exp_pscs_inh_dev,
                  Erev_exc_dev, Erev_inh_dev,
                  y_psns_dev, psn_weights_dev, exp_psns_dev, psn_times_dev,
psn_rates_dev, psn_seeds_dev,
                  spk_times_dev, neur_num_spks_dev, t, time_step, Nneur);
            cudaDeviceSynchronize();
            integrate_synapses<<<Ncon/SYN_BLOCK_SZ + 1, SYN_BLOCK_SZ>>>(
                  xs_dev, us_dev, exp_recs_dev, exp_facs_dev, Us_dev,
                  weights_dev, delays_dev, pre_syns_dev, post_syns_dev,
receptor_type_dev,
                  syn_num_spks_dev, neur_num_spks_dev, spk_times_dev,
                  AMPA_Amuont_dev, GABA_Amuont_dev,
                  t, Ncon, Nneur);
            cudaDeviceSynchronize();
      }
//    const char* error = cudaGetErrorString(cudaPeekAtLastError());
//    printf("%s\n", error);
//    error = cudaGetErrorString(cudaThreadSynchronize());
//    printf("%s\n", error);
      CUDA_CHECK_RETURN(
            cudaMemcpy(spk_times, spk_times_dev, sizeof(unsigned
int)*len_spk_tms, cudaMemcpyDeviceToHost));
      CUDA_CHECK_RETURN(
            cudaMemcpy(neur_num_spks, neur_num_spks_dev, sizeof(unsigned
int)*Nneur, cudaMemcpyDeviceToHost));
}


/*
 * kernel_api.h
 *
 *  Created on: 08 мая 2014 г.
 *      Author: pavel
 */

#ifndef KERNEL_API_H_
#define KERNEL_API_H_

namespace nnsim{

      void init_network(float h, int NumNeur, int NumConns, float SimTime);

      void init_neurs(float* a_arr, float* b1_arr, float* b2_arr, float* c_arr,
```

```
float* d_arr, float* k_arr,
                float* Cm_arr, float* Erev_exc_arr, float* Erev_inh_arr,
float* Ie_arr, float* Isyn_arr, float* tau_psc_exc_arr, float* tau_psc_inh_arr,
                float* Um_arr, float* Vm_arr, float* Vpeak_arr, float* Vr_arr,
float* Vt_arr, float* p1_arr, float* p2_arr);

      void init_synapses(float* tau_rec_arr, float* tau_fac_arr, float* U_arr,
                float* x_arr, float* y_arr, float* u_arr, float* weights_arr,
float* delays_arr,
                unsigned int* pre_conns_arr, unsigned int* post_conns_arr,
unsigned int* receptor_type_arr);

      void init_spikes(unsigned int* spike_times, unsigned int* neur_num_spikes,
                unsigned int* syn_num_spikes, unsigned int spk_times_len);

      int simulate(int useGPU);

      void init_recorder(unsigned int neur_num, unsigned int* neurs, unsigned
int con_num, unsigned int* conns);

      void get_neur_results(float* &Vm_res, float* &Um_res, float* &Isyn_res,
float* &y_exc_res, float* &y_inh_res, unsigned int &N);

      void get_conn_results(float* &x_res, float* &u_res, unsigned int &N);

      void get_mean_neur_results(float* &Vm_res, float* &Um_res, float*
&Isyn_res, float* &y_exc_res, float* &y_inh_res, unsigned int &N);

      void get_mean_conn_results(float* &x_res, float* &u_res, unsigned int &N);

      void get_spike_times(unsigned int* &spike_times, unsigned int*
&num_spikes_on_neur);

      void init_poisson(unsigned int* seeds, float* rates, float* weights, float
psn_tau);

      void init_mean_recorder(unsigned int num_pop_neur, unsigned int
num_pop_conn);

      void add_neur_mean_record(unsigned int pop_size, unsigned int* pop_neurs);

      void add_conn_mean_record(unsigned int pop_size, unsigned int* pop_conns);

}

#endif /* KERNEL_API_H_ */


/*
 * kernel_declatations.h
 *
 *  Created on: 14 апр. 2014 г.
 *      Author: pavel
 */

#ifndef KERNEL_DECLARATIONS_H_
#define KERNEL_DECLARATIONS_H_

#include "kernel_api.h"
#include "nnsim_constants.h"

namespace nnsim{
```

```
// step size in ms
float time_step;
int Ncon;
int Nneur;
unsigned int Tsim;

// Neural variables and parameters
float* Vms;
float* Ums;
float* Ies;
float* as;
float* b1_s;
float* b2_s;
float* cs;
float* ds;
float* ks;
float* Cms;
float* Vrs;
float* Vts;
float* Vpeaks;
float* p1_s;
float* p2_s;
float* Isyns;
float* Erev_exc;
float* Erev_inh;
float* AMPA_Amuont;
float* GABA_Amuont;
float* exp_pscs_exc;
float* exp_pscs_inh;

unsigned int* psn_times;
unsigned int* psn_seeds;
float* psn_rates;
float* y_psns;
float* exp_psns;
float* psn_weights;

unsigned int* spk_times;
unsigned int* neur_num_spks;
unsigned int* syn_num_spks;
unsigned int len_spk_tms;

// Synaptic parameters and variables
float* xs;
float* us;
float* Us;
float* exp_recs;
float* exp_facs;

float* weights;
unsigned int* delays;
unsigned int* pre_syns;
unsigned int* post_syns;
unsigned int* receptor_type;

// Variables for recording
unsigned int recorded_neur_num = 0;
unsigned int* neurs_to_record;
float* Vm_recorded;
float* Um_recorded;
float* Isyn_recorded;
float* y_exc_recorded;
float* y_inh_recorded;
```

```
	unsigned int recorded_con_num = 0;
	unsigned int* conns_to_record;
	float* x_recorded;
	float* u_recorded;

	unsigned int NumPopNeur = 0; // number of populations (neurons)
	unsigned int NumPopConn = 0; // number of populations (synapses)
	unsigned int CurrentPopNeur = 0;
	unsigned int CurrentPopConn = 0;
	unsigned int* PopNeurSizes;   // number of neuron in each population
	unsigned int* PopConnSizes;   // number of synapses in each population
	unsigned int** PopNeurs;            // neuron indices of each population
(size = max(PopNeurSizes)*NumPopNeur)
	unsigned int** PopConns;            // connection indices of each
population (size = max(PopConnSizes)*NumPopConn)

	float* Vm_means;
	float* Um_means;
	float* Isyn_means;
	float* y_exc_means;
	float* y_inh_means;

	float* x_means;
	float* u_means;
}

#endif /* KERNEL_DECLARATIONS_H_ */


# -*- coding: utf-8
# nnsim.py
'''
Created on 13 мая 2014 г.

@author: pavel
'''

import nnsim_pykernel
import numpy as np
np.random.seed(seed=0)

MeanSpkPeriod = 5.

psn_tau = 3.

neur_param = {}

neur_param['exc'] = {'a': 0.02, 'b_1': 0.5, 'b_2': 0.5, 'c': -40., 'd': 100.,
'k': 0.5, 'Cm': 50.,
                     'Vr': -60., 'Vt': -45., 'Vpeak': 40., 'p_1': 1., 'p_2':
1., 'Vm': -60., 'Um': 0.,
                     'Erev_AMPA': 0., 'Erev_GABA': -70., 'Isyn': 0.,
'tau_psc_exc': 3., 'tau_psc_inh': 7., 'Ie': 0.,
                     'psn_seed': None, 'psn_rate': 0., 'psn_weight': 1.}

neur_param['inh'] = {'a': 0.03, 'b_1': -2.0, 'b_2': -2.0, 'c': -50., 'd': 100.,
'k': 0.7, 'Cm': 100.,
                     'Vr': -60., 'Vt': -40., 'Vpeak': 35., 'p_1': 1., 'p_2':
1., 'Vm': -60., 'Um': 0.,
                     'Erev_AMPA': 0., 'Erev_GABA': -70., 'Isyn': 0.,
'tau_psc_exc': 3., 'tau_psc_inh': 7., 'Ie': 0.,
                     'psn_seed': None, 'psn_rate': 0., 'psn_weight': 1.}
```

```
syn_param = {}

syn_param['exc'] = {'tau_rec': 800., 'tau_fac': 0.00001,
                    'U': 0.5, 'receptor_type': 1}

syn_param['inh'] = {'tau_rec': 100., 'tau_fac': 1000.,
                    'U': 0.04, 'receptor_type': 2}

syn_default = {'y': 0., 'x': 1., 'u': 0., 'weight': 1., 'delay': 0.}

neur_arr = {'a': [], 'b_1': [], 'b_2': [], 'c': [], 'd': [], 'k': [], 'Cm': [],
                    'Vr': [], 'Vt': [], 'Vpeak': [], 'p_1': [], 'p_2': [],
'Vm': [], 'Um': [],
                    'Erev_AMPA': [], 'Erev_GABA': [], 'Isyn': [],
'tau_psc_exc': [], 'tau_psc_inh': [], 'Ie': [],
                    'psn_seed': [], 'psn_rate': [], 'psn_weight': []}

syn_arr = {'tau_rec': [], 'tau_fac': [], 'U': [],
                   'y': [], 'x': [], 'u': [], 'weight': [], 'delay': [],
                   'pre': [], 'post': [], 'receptor_type': []}
rec_from_neur = []
rec_from_syn = []

NumNodes = 0

NumConns = 0

def init():
    global NumNodes, NumConns
    NumNodes, NumConns = 0, 0
    global neur_arr, syn_arr, rec_from_neur, rec_from_syn
    neur_arr = {'a': [], 'b_1': [], 'b_2': [], 'c': [], 'd': [], 'k': [], 'Cm':
[],
                    'Vr': [], 'Vt': [], 'Vpeak': [], 'p_1': [], 'p_2': [],
'Vm': [], 'Um': [],
                    'Erev_AMPA': [], 'Erev_GABA': [], 'Isyn': [],
'tau_psc_exc': [], 'tau_psc_inh': [], 'Ie': [],
                    'psn_seed': [], 'psn_rate': [], 'psn_weight': []}

    syn_arr = {'tau_rec': [], 'tau_fac': [], 'U': [],
                   'y': [], 'x': [], 'u': [], 'weight': [], 'delay': [],
                   'pre': [], 'post': [], 'receptor_type': []}
    rec_from_neur = []
    rec_from_syn = []

def check_type(arg, ar_type=int):
    if type(arg) == list:
        for i in arg:
            if type(i) != ar_type:
                raise RuntimeError("Argument must be " + str(ar_type) + "or list
of " + str(ar_type))
        return arg
    elif type(arg) == np.ndarray:
        if arg.dtype == np.int:
            return arg
    elif type(arg) != ar_type:
        raise RuntimeError("Argument must be " + str(ar_type) + "or list of " +
str(ar_type))
    return [arg]

def create(N, n_type="exc", **kwargs):
    global neur_arr, NumNodes
```

```
    default_params=neur_param[n_type].copy()

    for key, value in kwargs.items():
        if type(value) in [list, tuple, np.ndarray]:
            neur_arr[key].extend(value[:N])
        elif type(value) not in [str, dict]:
            neur_arr[key].extend([value]*N)
        elif type(value) == dict:
            if value['distr'] == 'normal':
                std = value['std']
                mean = value['mean']
                if value.get('abs', True) == True:
                    neur_arr[key].extend(np.abs(mean + std*np.random.randn(N)))
                else:
                    neur_arr[key].extend(mean + std*np.random.randn(N))
            elif value['distr'] == 'uniform':
                low = value['low']
                high = value['high']
                neur_arr[key].extend(np.random.uniform(low, high, size=N))
            elif value['distr'] == 'gamma':
                shape = value['shape']
                scale = value['scale']
                loc = value['loc']
                neur_arr[key].extend(loc + np.random.gamma(shape, scale,
size=N))
        else:
            raise RuntimeError("{0} must be a number or dict".format(key))
        default_params.pop(key)

    for key, value in default_params.items():
            neur_arr[key].extend([value]*N)
    NumNodes += N
    return [i for i in xrange(NumNodes - N, NumNodes)]

def set_nparam(n_idx, **kwargs):
    if type(n_idx) in [list, np.ndarray]:
        n_idx = n_idx[0]
    for key, value in kwargs.items():
        neur_arr[key][n_idx] = value

def connect(pre, post, conn_spec='one_to_one', syn='exc', **kwargs):
    global syn_arr, NumConns
    pre = check_type(pre)
    post = check_type(post)
    pre_ext = []
    post_ext = []
    syn_ext ={}
    if(conn_spec == 'one_to_one'):
        if (len(pre) != len(post)):
            raise RuntimeError("Lengths of pre and post must be equal")
        pre_ext = pre
        post_ext = post
    elif (conn_spec == 'all_to_all'):
        for i in pre:
            pre_ext.extend([i]*len(post))
            post_ext.extend(post)
    elif type(conn_spec) == dict:
        if conn_spec['rule'] == 'fixed_total_num':
            for i in xrange(conn_spec['N']):
                pre_ext.append(pre[np.random.randint(len(pre))])
                post_ext.append(post[np.random.randint(len(post))])
        if conn_spec['rule'] == 'fixed_outdegree':
            for i in pre:
```

```
                n_post = conn_spec['N']
                pre_ext.extend([i]*n_post)
                post_ext.extend(np.random.permutation(post)[:n_post])
        if conn_spec['rule'] == 'mean_outdegree':
            for i in pre:
                n_post = np.int(np.abs(conn_spec['N_mean'] +
conn_spec['N_std']*np.random.randn()))
                pre_ext.extend([i]*n_post)
                post_ext.extend(np.random.permutation(post)[:n_post])
    else:
        raise RuntimeError("conn_spec must be one_to_one or all_to_all or dict")

    for key, value in syn_param[syn].items() + syn_default.items():
        syn_ext[key] = [value]*len(pre_ext)

    for key, value in kwargs.items():
        if type(value) not in [str, list, tuple, dict, np.ndarray]:
            syn_ext[key] = [value]*len(pre_ext)
        elif type(value) == dict:
            if value['distr'] == 'normal':
                std = value['std']
                mean = value['mean']
                if value.get('abs', True) == True:
                    syn_ext[key] = np.abs(mean +
std*np.random.randn(len(pre_ext)))
                else:
                    syn_ext[key] = mean + std*np.random.randn(len(pre_ext))
            elif value['distr'] == 'uniform':
                low = value['low']
                high = value['high']
                syn_ext[key] = np.random.uniform(low, high, size=len(pre_ext))
            elif value['distr'] == 'gamma':
                shape = value['shape']
                scale = value['scale']
                loc = value['loc']
                syn_ext[key] = loc + np.random.gamma(shape, scale,
size=len(pre_ext))
        else:
            raise RuntimeError("{0} must be a number or dict".format(key))
    syn_ext['pre'] = pre_ext
    syn_ext['post'] = post_ext
    for key, value in syn_ext.items():
        syn_arr[key].extend(value)

    NumConns += len(pre_ext)
    return [i for i in xrange(NumConns - len(pre_ext), NumConns)]

def record(nodes, node_type='neur'):
    global rec_from_neur, rec_from_syn
    if node_type == 'neur':
        rec_from_neur.extend(check_type(nodes))
    elif node_type == 'syn':
        rec_from_syn.extend(check_type(nodes))
    print rec_from_neur

pop_idx = {'neur': 0, 'syn': 0}
pop_nodes = {'neur': [], 'syn': []}
pop_names = {'neur': [], 'syn': []}

def mean_record(nodes, node_type='neur', name=None):
    pop_nodes[ntype].append(nodes)
    if name == None:
        name = pop_idx[ntype]
```

```
    pop_names[ntype].append(name)
    pop_idx[ntype] += 1
    pop_nodes[ntype].append(nodes)

def get_results(mean=False):
    if mean:
        num_neur_rec = pop_idx['neur']
        num_syn_rec = pop_idx['syn']
        mean = 1
    else:
        num_neur_rec = len(rec_from_neur)
        num_syn_rec = len(rec_from_syn)
        mean = 0
    (Vm_, Um_, Isyn_, y_exc_, y_inh_, x_, u_) = nnsim_pykernel.get_results(mean)
    Vm = []
    Um = []
    Isyn = []
    y_exc = []
    y_inh = []
    x = []
    u = []
    if len(Vm_) == 0:
        return (Vm, Um, Isyn, y_exc, y_inh, x, u)

    start = 0
    Tsim = len(Vm_)/num_neur_rec
    stop = Tsim
    for i in xrange(num_neur_rec):
        Vm.append(Vm_[start:stop])
        Um.append(Um_[start:stop])
        Isyn.append(Isyn_[start:stop])
        y_exc.append(y_exc_[start:stop])
        y_inh.append(y_inh_[start:stop])
        stop += Tsim
        start += Tsim

    if len(x_) == 0:
        return (Vm, Um, Isyn, y_exc, y_inh, x, u)

    start = 0
    Tsim = len(x_)/num_syn_rec
    stop = Tsim
    for i in xrange(num_syn_rec):
        x.append(x_[start:stop])
        u.append(u_[start:stop])
        stop += Tsim
        start += Tsim

    return (Vm, Um, Isyn, y_exc, y_inh, x, u)

def get_spk_times():
    global spk_times, n_spike
    (spk_times, n_spike) = nnsim_pykernel.get_spk_times()

    spikes = []
    for i in xrange(NumNodes):
        spikes.append([spk_times[NumNodes*sn + i]*tm_step for sn in
xrange(n_spike[i])])
    return spikes

def get_ordered_spikes():
    return order_spikes(get_spk_times())
```

```
def order_spikes(spikes):
    times = []
    senders = []
    for i in xrange(NumNodes):
        times.extend(spikes[i])
        senders.extend([i]*len(spikes[i]))
    return (times, senders)

def simulate(h, SimTime, gpu=False):
    global tm_step
    tm_step = h
    nnsim_pykernel.init_network(h, NumNodes, NumConns, SimTime)
    psn_keys = ['psn_seed', 'psn_rate', 'psn_weight']

    args = {}
    for key, val in neur_arr.items():
        if key not in psn_keys:
            args[key] = np.array(val, dtype='float32')
    nnsim_pykernel.init_neurs(**args)

    psn_args = {}
    psn_args['psn_seed'] = np.array(np.random.randint(2147483647,
size=NumNodes), dtype='uint32')
    for i in xrange(NumNodes):
        if neur_arr['psn_seed'][i] != None:
            psn_args['psn_seed'][i] = neur_arr['psn_seed'][i]

    psn_args['psn_rate'] = np.array(neur_arr['psn_rate'], dtype='float32')
    psn_args['psn_weight'] = np.array(neur_arr['psn_weight'], dtype='float32')
    psn_args['psn_tau'] = psn_tau
    nnsim_pykernel.init_poisson(**psn_args)

    args = {}
    for key, val in syn_arr.items():
        args[key] = np.array(val, dtype='float32')
    for key in ['pre', 'post', 'receptor_type']:
        args[key] = np.array(syn_arr[key], dtype='uint32')
    nnsim_pykernel.init_synapses(**args)

    args = {}
    args['sps_times'] = np.zeros(NumNodes*SimTime/MeanSpkPeriod, dtype='uint32')
    args['neur_num_spk'] = np.zeros(NumNodes, dtype='uint32')
    args['syn_num_spk'] = np.zeros(NumConns, dtype='uint32')
    nnsim_pykernel.init_spikes(**args)

    nnsim_pykernel.init_recorder(len(rec_from_neur), rec_from_neur,
                                 len(rec_from_syn), rec_from_syn)

    nnsim_pykernel.init_mean_recorder(pop_idx['neur'], pop_idx['syn'])
    for i in pop_nodes['neur']:
        nnsim_pykernel.add_neur_mean_record(np.array(i, dtype='uint32'))

    for i in pop_nodes['syn']:
        nnsim_pykernel.add_conn_mean_record(np.array(i, dtype='uint32'))
    gpu = [0, 1][gpu]
        #gpu = 1
    #else:
        #gpu = 0

    nnsim_pykernel.simulate(gpu)

print "  --NNSIM--  "
```

```
/*
 * nnsim_constants.h
 *
 *  Created on: 05.11.2014
 *      Author: pavel
 */

#ifndef NNSIM_CONSTANTS_H_
#define NNSIM_CONSTANTS_H_

#define AMPA_RECEPTOR 1
#define GABA_RECEPTOR 2

#define NEUR_BLOCK_SZ 256
#define SYN_BLOCK_SZ 512

#endif /* NNSIM_CONSTANTS_H_ */
```

```
/*
 * cuda_kernel_declarations.h
 *
 *  Created on: 05.11.2014
 *      Author: pavel
 */

#ifndef CUDA_KERNEL_DECLARATIONS_H_
#define CUDA_KERNEL_DECLARATIONS_H_

#include <stdio.h>

#define CUDA_CHECK_RETURN(value) {
                    \
      cudaError_t _m_cudaStat = value;
                    \
      if (_m_cudaStat != cudaSuccess) {
                    \
            fprintf(stderr, "Error %s at line %d in file %s\n",
            \
                        cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);
      \
            exit(1);
                              \
      } }

namespace nnsim {

      extern float time_step;
      extern int Ncon;
      extern int Nneur;
      extern unsigned int Tsim;

      // Neural variables and parameters
      extern float* Vms;
      extern float* Ums;
      extern float* Ies;
      extern float* as;
      extern float* b1_s;
      extern float* b2_s;
      extern float* cs;
```

```
        extern float* ds;
        extern float* ks;
        extern float* Cms;
        extern float* Vrs;
        extern float* Vts;
        extern float* Vpeaks;
        extern float* p1_s;
        extern float* p2_s;
        extern float* Isyns;
        extern float* Erev_exc;
        extern float* Erev_inh;
        extern float* AMPA_Amuont;
        extern float* GABA_Amuont;
        extern float* exp_pscs_exc;
        extern float* exp_pscs_inh;
        extern unsigned int* psn_times;
        extern unsigned int* psn_seeds;
        extern float* psn_rates;
        extern float* y_psns;
        extern float* exp_psns;
        extern float* psn_weights;
        extern unsigned int* spk_times;
        extern unsigned int* neur_num_spks;
        extern unsigned int* syn_num_spks;
        extern unsigned int len_spk_tms;

        // Synaptic parameters and variables
        extern float* xs;
        extern float* us;
        extern float* Us;
        extern float* exp_recs;
        extern float* exp_facs;
        extern float* weights;
        extern unsigned int* delays;
        extern unsigned int* pre_syns;
        extern unsigned int* post_syns;
        extern unsigned int* receptor_type;
}
        using namespace nnsim;

        // Neural variables and parameters
        float* Vms_dev;
        float* Ums_dev;
        float* Ies_dev;
        float* as_dev;
        float* b1_s_dev;
        float* b2_s_dev;
        float* cs_dev;
        float* ds_dev;
        float* ks_dev;
        float* Cms_dev;
        float* Vrs_dev;
        float* Vts_dev;
        float* Vpeaks_dev;
        float* p1_s_dev;
        float* p2_s_dev;
        float* Isyns_dev;
        float* Erev_exc_dev;
        float* Erev_inh_dev;
        float* AMPA_Amuont_dev;
        float* GABA_Amuont_dev;
        float* exp_pscs_exc_dev;
        float* exp_pscs_inh_dev;
```

```
	unsigned int* psn_times_dev;
	unsigned int* psn_seeds_dev;
	float* psn_rates_dev;
	float* y_psns_dev;
	float* exp_psns_dev;
	float* psn_weights_dev;
	unsigned int* spk_times_dev;
	unsigned int* neur_num_spks_dev;
	unsigned int* syn_num_spks_dev;

	// Synaptic parameters and variables
	float* xs_dev;
	float* us_dev;
	float* Us_dev;
	float* exp_recs_dev;
	float* exp_facs_dev;
	float* weights_dev;
	unsigned int* delays_dev;
	unsigned int* pre_syns_dev;
	unsigned int* post_syns_dev;
	unsigned int* receptor_type_dev;

	__host__ void init_mem(){
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Vms_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Ums_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Ies_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &as_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &b1_s_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &b2_s_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &cs_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &ds_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &ks_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Cms_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Vrs_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Vts_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Vpeaks_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &p1_s_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &p2_s_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Isyns_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Erev_exc_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &Erev_inh_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &AMPA_Amuont_dev,
sizeof(float)*Nneur));
		CUDA_CHECK_RETURN(cudaMalloc((void**) &GABA_Amuont_dev,
sizeof(float)*Nneur));
```

```
            CUDA_CHECK_RETURN(cudaMalloc((void**) &exp_pscs_exc_dev,
sizeof(float)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &exp_pscs_inh_dev,
sizeof(float)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &psn_rates_dev,
sizeof(float)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &y_psns_dev,
sizeof(float)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &exp_psns_dev,
sizeof(float)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &psn_weights_dev,
sizeof(float)*Nneur));

            CUDA_CHECK_RETURN(cudaMalloc((void**) &psn_times_dev,
sizeof(unsigned int)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &psn_seeds_dev,
sizeof(unsigned int)*Nneur));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &neur_num_spks_dev,
sizeof(unsigned int)*Nneur));

            CUDA_CHECK_RETURN(cudaMalloc((void**) &spk_times_dev,
sizeof(unsigned int)*len_spk_tms));

            CUDA_CHECK_RETURN(cudaMalloc((void**) &syn_num_spks_dev,
sizeof(unsigned int)*Ncon));

            CUDA_CHECK_RETURN(cudaMalloc((void**) &xs_dev, sizeof(float)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &us_dev, sizeof(float)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &Us_dev, sizeof(float)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &exp_recs_dev,
sizeof(float)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &exp_facs_dev,
sizeof(float)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &weights_dev,
sizeof(float)*Ncon));

            CUDA_CHECK_RETURN(cudaMalloc((void**) &delays_dev, sizeof(unsigned
int)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &pre_syns_dev, sizeof(unsigned
int)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &post_syns_dev,
sizeof(unsigned int)*Ncon));
            CUDA_CHECK_RETURN(cudaMalloc((void**) &receptor_type_dev,
sizeof(unsigned int)*Ncon));
      }

      __host__ void copy2device(){
            CUDA_CHECK_RETURN(cudaMemcpy(Vms_dev, Vms, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Ums_dev, Ums, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Ies_dev, Ies, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(as_dev, as, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(b1_s_dev, b1_s, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(b2_s_dev, b2_s, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(cs_dev, cs, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(ds_dev, ds, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
```

```
            CUDA_CHECK_RETURN(cudaMemcpy(ks_dev, ks, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Cms_dev, Cms, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Vrs_dev, Vrs, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Vts_dev, Vts, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Vpeaks_dev, Vpeaks,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(p1_s_dev, p1_s, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(p2_s_dev, p2_s, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Isyns_dev, Isyns, sizeof(float)*Nneur,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Erev_exc_dev, Erev_exc,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Erev_inh_dev, Erev_inh,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(AMPA_Amuont_dev, AMPA_Amuont,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(GABA_Amuont_dev, GABA_Amuont,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(exp_pscs_exc_dev, exp_pscs_exc,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(exp_pscs_inh_dev, exp_pscs_inh,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(psn_rates_dev, psn_rates,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(y_psns_dev, y_psns,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(exp_psns_dev, exp_psns,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(psn_weights_dev, psn_weights,
sizeof(float)*Nneur, cudaMemcpyHostToDevice));

            CUDA_CHECK_RETURN(cudaMemcpy(psn_times_dev, psn_times,
sizeof(unsigned int)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(psn_seeds_dev, psn_seeds,
sizeof(unsigned int)*Nneur, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(neur_num_spks_dev, neur_num_spks,
sizeof(unsigned int)*Nneur, cudaMemcpyHostToDevice));

            CUDA_CHECK_RETURN(cudaMemcpy(spk_times_dev, spk_times,
sizeof(unsigned int)*len_spk_tms, cudaMemcpyHostToDevice));

            CUDA_CHECK_RETURN(cudaMemcpy(syn_num_spks_dev, syn_num_spks,
sizeof(unsigned int)*Ncon, cudaMemcpyHostToDevice));

            CUDA_CHECK_RETURN(cudaMemcpy(xs_dev, xs, sizeof(float)*Ncon,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(us_dev, us, sizeof(float)*Ncon,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(Us_dev, Us, sizeof(float)*Ncon,
cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(exp_recs_dev, exp_recs,
sizeof(float)*Ncon, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(exp_facs_dev, exp_facs,
sizeof(float)*Ncon, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(weights_dev, weights,
sizeof(float)*Ncon, cudaMemcpyHostToDevice));
```

```
            CUDA_CHECK_RETURN(cudaMemcpy(delays_dev, delays, sizeof(unsigned
int)*Ncon, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(pre_syns_dev, pre_syns, sizeof(unsigned
int)*Ncon, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(post_syns_dev, post_syns,
sizeof(unsigned int)*Ncon, cudaMemcpyHostToDevice));
            CUDA_CHECK_RETURN(cudaMemcpy(receptor_type_dev, receptor_type,
sizeof(unsigned int)*Ncon, cudaMemcpyHostToDevice));
      }


#endif /* CUDA_KERNEL_DECLARATIONS_H_ */


/*
 * cuda_kernel_api.h
 *
 *  Created on: 08.11.2014
 *      Author: pavel
 */

#ifndef CUDA_KERNEL_API_H_
#define CUDA_KERNEL_API_H_

void simulateOnGpu();


#endif /* CUDA_KERNEL_API_H_ */


/*
 * cuda_kernel_api.cpp
 *
 *  Created on: 01 нояб. 2014 г.
 *      Author: pavel
 */

#include "cuda_kernel_declarations.h"
#include "nnsim_constants.h"

__device__ float get_random(unsigned int *seed){
      // return random number homogeneously distributed in interval [0:1]
      unsigned long a = 16807;
      unsigned long m = 2147483647;
      unsigned long x = (unsigned long) *seed;
      x = (a * x) % m;
      *seed = (unsigned int) x;
      return ((float)x)/m;
}

__global__ void integrate_synapses(float* x, float* u, float* exp_rec, float*
exp_fac, float* U,
                                                      float* weight, unsigned int*
delay, unsigned int* pre_syn, unsigned int* post_syn, unsigned int*
receptor_type,
                                                      unsigned int* syn_num_spk,
unsigned int* neur_num_spk, unsigned int* spk_time,
                                                      float* AMPA_Amuont, float*
GABA_Amuont,
                                                      unsigned int t, int Ncon, int
Nneur){
      unsigned int c = blockDim.x*blockIdx.x + threadIdx.x;
      if (c < Ncon){
```

```
            x[c] = (x[c] - weight[c])*exp_rec[c] + weight[c];
            u[c] = u[c]*exp_fac[c];

            if (syn_num_spk[c] < neur_num_spk[pre_syn[c]]){
                  if (t >= delay[c] && spk_time[Nneur*syn_num_spk[c] +
pre_syn[c]] == t - delay[c]){
                        u[c] += U[c]*(1.0f - u[c]);
                  float delta_x = x[c]*u[c];
                        x[c] -= delta_x;
                        syn_num_spk[c]++;

                        if (receptor_type[c] == AMPA_RECEPTOR){
                              atomicAdd(&AMPA_Amuont[post_syn[c]], delta_x);
                        } else if (receptor_type[c] == GABA_RECEPTOR){
                              atomicAdd(&GABA_Amuont[post_syn[c]], delta_x);
                        }
                  }
            }

      }
}

__device__ __inline__ float check_pow(float x, float degr){
      if (degr == 1.0f){
            return x;
      } else {
            return powf(x, degr);
      }
}

__global__ void integrate_neurons(float* Vms, float* Ums,
            float* a, float* b1, float* b2, float* c, float* d, float* k, float*
p1, float* p2,
            float* Vpeak, float* Vr, float* Vt, float* Cm, float* Ie, float*
Isyn,
            float* AMPA_Amuont, float* GABA_Amuont, float* exp_pscs_exc, float*
exp_pscs_inh,
            float* Erev_exc, float* Erev_inh,
            float* y_psn, float* psn_weight, float* exp_psn, unsigned int*
psn_time, float* psn_rate, unsigned int* psn_seed,
            unsigned int* spk_time, unsigned int* neur_num_spk,
            unsigned int t, float time_step, int Nneur){
      unsigned int n = blockIdx.x*blockDim.x + threadIdx.x;
      float v1, u1, v2, u2, v3, u3, v4, u4;
      if (n < Nneur){
            y_psn[n] *= exp_psn[n];
            while (psn_time[n] == t){
                  y_psn[n] += psn_weight[n];
                  psn_time[n] -= -1 +
(1000.0f/(time_step*psn_rate[n]))*log(get_random(psn_seed + n));
            }

            AMPA_Amuont[n] *= exp_pscs_exc[n];
            GABA_Amuont[n] *= exp_pscs_inh[n];

            float Vm = Vms[n];
            float Um = Ums[n];
            float Isyn_new = -(AMPA_Amuont[n] + y_psn[n])*(Vm - Erev_exc[n]) -
GABA_Amuont[n]*(Vm - Erev_inh[n]);

            v1 = (k[n]*(Vm - Vr[n])*(Vm - Vt[n]) - Um + Ie[n] +
Isyn[n])*time_step/Cm[n];
            u1 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vm - Vr[n]),
```

```
p1[n]) - Um : b2[n]*check_pow((Vm - Vr[n]), p2[n]) - Um);
            Vms[n] = Vm + v1*0.5f;
            Ums[n] = Um + u1*0.5f;
            v2 = (k[n]*(Vms[n] - Vr[n])*(Vms[n] - Vt[n]) - Ums[n] + Ie[n] +
(Isyn_new + Isyn[n])*0.5f)*time_step/Cm[n];
            u2 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vms[n] -
Vr[n]), p1[n]) - Ums[n] : b2[n]*check_pow((Vms[n] - Vr[n]), p2[n]) - Ums[n]);
            Vms[n] = Vm + v2*0.5f;
            Ums[n] = Um + u2*0.5f;
            v3 = (k[n]*(Vms[n] - Vr[n])*(Vms[n] - Vt[n]) - Ums[n] + Ie[n] +
(Isyn_new + Isyn[n])*0.5f)*time_step/Cm[n];
            u3 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vms[n] -
Vr[n]), p1[n]) - Ums[n] : b2[n]*check_pow((Vms[n] - Vr[n]), p2[n]) - Ums[n]);
            Vms[n] = Vm + v3;
            Ums[n] = Um + u3;
            v4 = (k[n]*(Vms[n] - Vr[n])*(Vms[n] - Vt[n]) - Ums[n] + Ie[n] +
Isyn_new)*time_step/Cm[n];
            u4 = time_step*a[n]*(Vms[n] < Vr[n] ? b1[n]*check_pow((Vms[n] -
Vr[n]), p1[n]) - Ums[n] : b2[n]*check_pow((Vms[n] - Vr[n]), p2[n]) - Ums[n]);
            Vms[n] = Vm + (v1 + 2.0f*(v2 + v3) + v4)*0.16666666f;
            Ums[n] = Um + (u1 + 2.0f*(u2 + u3) + u4)*0.16666666f;

            if (Vm > Vpeak[n]){
                  spk_time[Nneur*neur_num_spk[n] + n] = t;
                  neur_num_spk[n]++;
                  Vms[n] = c[n];
                  Ums[n] = Um + d[n];
            }
            Isyn[n] = Isyn_new;
      }
}

void simulateOnGpu(){
      init_mem();
      copy2device();
      for (unsigned int t = 0; t < Tsim; t++){
            integrate_neurons<<<Nneur/NEUR_BLOCK_SZ + 1, NEUR_BLOCK_SZ>>>(
                  Vms_dev, Ums_dev, as_dev, b1_s_dev, b2_s_dev, cs_dev, ds_dev,
ks_dev, p1_s_dev, p2_s_dev,
                  Vpeaks_dev, Vrs_dev, Vts_dev, Cms_dev, Ies_dev, Isyns_dev,
                  AMPA_Amuont_dev, GABA_Amuont_dev, exp_pscs_exc_dev,
exp_pscs_inh_dev,
                  Erev_exc_dev, Erev_inh_dev,
                  y_psns_dev, psn_weights_dev, exp_psns_dev, psn_times_dev,
psn_rates_dev, psn_seeds_dev,
                  spk_times_dev, neur_num_spks_dev, t, time_step, Nneur);
            cudaDeviceSynchronize();
            integrate_synapses<<<Ncon/SYN_BLOCK_SZ + 1, SYN_BLOCK_SZ>>>(
                  xs_dev, us_dev, exp_recs_dev, exp_facs_dev, Us_dev,
                  weights_dev, delays_dev, pre_syns_dev, post_syns_dev,
receptor_type_dev,
                  syn_num_spks_dev, neur_num_spks_dev, spk_times_dev,
                  AMPA_Amuont_dev, GABA_Amuont_dev,
                  t, Ncon, Nneur);
            cudaDeviceSynchronize();
      }
      CUDA_CHECK_RETURN(
            cudaMemcpy(spk_times, spk_times_dev, sizeof(unsigned
int)*len_spk_tms, cudaMemcpyDeviceToHost));
      CUDA_CHECK_RETURN(
            cudaMemcpy(neur_num_spks, neur_num_spks_dev, sizeof(unsigned
int)*Nneur, cudaMemcpyDeviceToHost));
}
```

Контрольные вопросы и задания

1. Какова роль математического моделирования в нейронауке?

2. Какие существуют стандартные средства для моделирования спайковых нейронных сетей?

3. Какие преимущества предоставляют GPGPU вычисления?

4. Назовите основные типы памяти видеопроцессора доступные программисту при написании CUDA приложений.

5. Соберите спайковую нейронную сеть воспроизводящую явление кратковременной пластичности и выведите динамику синаптических переменных.

6. Соберите нейронную сеть состоящую из 2-х нейронов, связанных однонаправленной возбуждающей связью от нейрона который постоянно генерирует спайки ко второму тормозному нейрону. И меняя вес связи найдите пороговое значение при котором любой спайк на генераторе вызывает спайк на постсинаптическом нейроне.

7. Соберите такую же сеть как в предыдущем задании, только на этот раз пусть постсинаптический нейрон будет тормозным. Задание такое же.

8. Соберите такую же сеть как в 6-м задании. Только на этот раз меняйте не вес связи, а время затухания постсинаптического тока, при зафиксированном весе. И найдите пороговое значение времени затухания постсинаптического тока при котором каждый спайк на нейроне-генераторе вызывает спайк на постсинапсе. Найдите соотношение на сколько нужно увеличить время затухания, чтобы скомпенсировать уменьшение веса в 1.5, 2 и 3 раза.

9. Соберите нейронную сеть состоящую из 1000 нейронов и просимулируйте её динамику в одном случае без использования GPGPU ускорения, а в другом с использованием. Сравните времена расчёта.

Список использованной литературы

1. Hodgkin, A. L.; Huxley, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve // The Journal of physiology 1952. 117 (4), 500–544.
2. Izhikevich E.M. Simple model of spiking neurons. // IEEE Trans. Neural Netw. 2003. Vol. 14, № 6. P. 1569–1572.
3. Tsodyks M., Markram H. The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability // PNAS. 1997. Vol. 94. P. 719–723.
4. Wagenaar D.A., Pine J., Potter S.M. An extremely rich repertoire of bursting patterns during the development of cortical cultures. // BMC Neurosci. 2006. Vol. 7. P. 11.
5. Pimashkin A. et al. Spiking signatures of spontaneous activity bursts in hippocampal cultures. // Front. Comput. Neurosci. 2011. Vol. 5. P. 46.
6. А.В. Боресков, А.А. Харламов. Основы работы с технологией CUDA. // Издательство "ДМК Пресс", 2010

Ссылки из интернета:

1. http://www.ixbt.com/video3/cuda-1.shtml
2. http://www.nvidia.ru/object/cuda-parallel-computing-ru.html
3. NVIDIA CUDA 4.0 Programming Guide http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ GPGPU ВЫЧИСЛЕНИЯ ДЛЯ МОДЕЛИРОВАНИЯ ДИНАМИКИ СПАЙКОВЫХ НЕЙРОННЫХ СЕТЕЙ

Авторы:
Павел Михайлович **Есир**, Александр Юрьевич **Симонов**

Учебно-методическое пособие

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«Нижегородский государственный университет им. Н.И. Лобачевского».

603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать . Формат 60x84 1/16.
Бумага офсетная. Печать офсетная. Гарнитура Таймс.
Усл. печ. л. . Уч.-изд. л. .
Заказ № . Тираж 70 экз.

Отпечатано в типографии Нижегородского госуниверситета
им. Н.И. Лобачевского
603600, г. Нижний Новгород, ул. Большая Покровская, 37
Лицензия ПД № 18-0099 от 14.05.01