Ускорение процесса обучения нейросети за счет использования графического акселератора

А.А.Лукьяница, Б.Г.Севрюков

Цель настоящей работы – исследование возможности эффективного применения процессоров видеокарт для ускорения вычислений, связанных с обучением нейронных сетей. Так как видеокарты, по сути – многопроцессорные системы, обладающие несколькими уровнями параллелизма, необходимой частью работы стало изменение традиционной схемы вычислений в соответствии с принципами параллельного программирования. В ходе исследования рассматривалась модель трехслойного персептрона с сигмоидной активационной функцией. Обучение нейросети происходило в соответствии с алгоритмом обратного распространения ошибки. Однако традиционная схема вычислений была адаптирована с учетом особенностей параллелизма видеокарты. С целью уменьшения числа обменов с глобальной памятью видеокарты и увеличения отношения объема вычислений к объему загружаемых из нее данных все массивы данных должны быть записанными в двумерные массивы и обрабатываться на двумерной решетке. Для этого мы использовали пакетную обработку векторов обучающей выборки, а также изменили порядок вычислений. В разработанном нами алгоритме схема вычислений преобразована таким образом, чтобы во время обработки данных оперировать блоками нужного размера и обеспечить загруженность процессорных элементов, способную скрыть задержку при доступе к глобальной памяти видеокарты. Это также позволит ускорить обработку за счет такой особенности параллелизма видеокарты, как векторность процессорного элемента графической карты. Использование указанных особенностей параллелизма видеокарты позволило на два порядка ускорить процесс обучения нейронной сети.

1. Введение

Идея использовать процессоры видеокарт (далее - GPU) не только для вывода графики, но и для вычислений общего назначения (GPGPU), возникла давно, с выходом на рынок самых первых программируемых видеоакселераторов [2]. С увеличением гибкости этих чипов в плане программирования и ростом их вычислительной мощности (практически двукратный ежегодный рост производительности), эти идеи получили свое воплощение в многочисленных реальных проектах [3], а высокая производительность на определенных классах задач привлекает все больше исследователей.

Целью настоящей работы является исследование возможности ускорения процесса настройки искусственной нейронной сети путем использования GPU. Выбор объекта исследования обусловлен как широкой областью применения нейронных сетей, так и научной специализацией авторов. На принципиальную возможность ускорения процесса настройки нейросети наталкивает то обстоятельство, что нейросеть прямого распространения, которую часто называют персептроном [1], по своей сути является параллельной системой. Это связано с тем, что нейроны в пределах каждого слоя не взаимодействуют друг с другом, и поэтому процессы обработки сигнала каждым нейроном одного слоя могут моделироваться параллельно. Однако стандартную схему расчета нейросети необходимо адаптировать с учетом особенностей GPU, на которых мы остановимся позже.

В отличие от современных стандартных процессоров массового использования (далее – CPU), GPU предлагают свыше 24-х (128 – на более дорогих моделях, например GeForce 8800 GTX) арифметико-логических устройств (ALU) и сотни аппаратных нитей, способных выполнять один и тот же код параллельно. Дополнительно параллелизм усиливается за счет использования набора векторных команд. Причем, благодаря огромному рынку видеоигр (миллиарды долларов ежегодно), стоимость вычислений на видеокартах значительно ниже, чем на CPU.

В силу своей специфики GPU, предназначенные для ускорения обработки потоков информации о графическом представлении сцены (см. Рис. 1), обладают параллелизмом по данным. В таком контексте ядрами потоковой обработки являются специальные программы – «шейдеры», загружаемые в память видеокарты. В целях простоты изложения будем считать эквивалентными понятия: «поток», «массив данных», «текстура» [4].

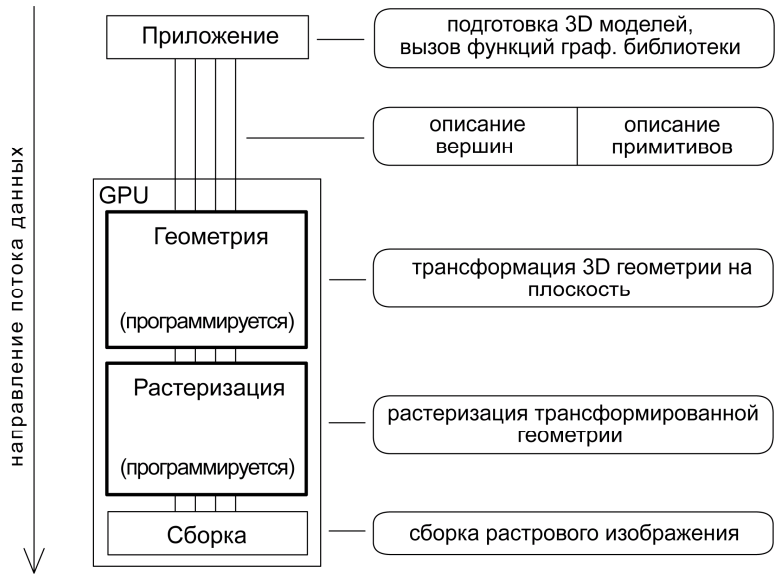

Рис. 1. Базовая архитектура GPU.

Вообще говоря, в GPU можно выделить такие уровни параллелизма, как: суперскалярность процессорного элемента (ПЭ), векторность, множественность ПЭ и аппаратных нитей, запускаемых на них, параллелизм работы процессорных элементов и доступа к памяти, множественность графических процессоров (появились технологии, позволяющие объединять видеокарты в однородные вычислительную системы).

Основные особенности традиционных вычислений общего назначения через графический интерфейс GPU заключаются в следующем:

1. В этих рамках топология GPU как параллельной системы – двумерная решетка. Это значит, что данные, поступающие на обработку в GPU и получаемые в результате работы программы на GPU, лучше всего представлять в виде двумерного массива значений, размер которого, однако, ограничен аппаратным интерфейсом

2. GPU позволяют производить арифметические операции вычисление элементарных функций, производить другие операции над векторами данных. Эти операции обеспечивают одновременное выполнение операций над всеми элементами вектора.

3. Медленный обмен данными между видеокартой и CPU: необходимо по возможности уменьшать этот объем либо увеличить отношение объема вычислений к объему данных, а также максимально полно использовать кеширование загружаемых данных, так как доступ к кешированным данным дает ничтожную по сравнению с доступом к глобальной памяти видеокарты задержку.

4. Традиционная операция scatter - запись результатов обработки, производимой на процессорном элементе, в память, доступную для других потоков, - может приводить к значительной потере производительности (в отличие от операции gather – сбора данных)

Однако в современных GPU для вычислений общего назначения выделен аппаратный интерфейс. Это, например, технология CUDA[13], доступная на видеокартах NVIDIA GeForce серии 8 и выше. Появилась возможность управлять памятью разделяемой между аппаратными нитями, запущенными в рамках одного мультипроцессора. Каждая аппаратная нить имеет возможность чте-

ния и записи произвольных участков глобальной памяти, то есть реализация gather и scatter ограничена только латентностью видеопамяти. Аппаратный интерфейс позволяет оперировать не только двумерными, но также трехмерной и линейной решетками произвольного размера. Среди прочих преимуществ этого интерфейса можно выделить увеличение скорости как загрузки данных в GPU, так и скачивания результатов обработки.

2. Модель нейросети

В качестве объекта исследования была рассмотрена трехслойная нейронная сеть с сигмоидной активационной функцией. Настройка производилась в соответствии с алгоритмом обратного распространения ошибки [1].

Остановимся подробнее на модели. Мы рассматриваем трехслойную нейронную сеть с нейронами Мак-Каллока-Питтса, архитектура которой приведена на Рис. 2. Пусть входной слой содержит m нейронов, скрытый – n, выходной – k. Тогда связи между входным и скрытым слоями характеризуются матрицей весов $W^1_{M\times N}$, связи между скрытым и выходным слоями – матрицей $W^2_{N\times K}$.

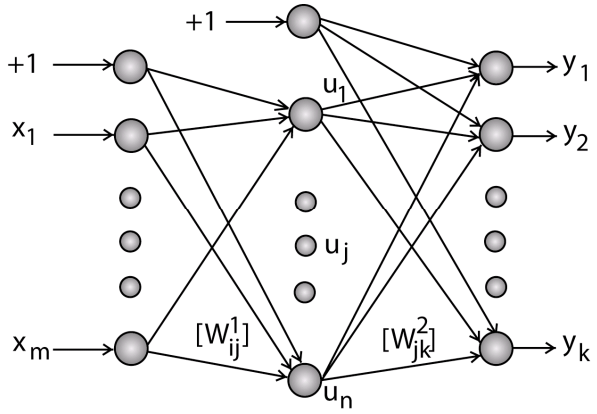

Рис. 2. Архитектура трехслойной нейронной сети.

В качестве активационной функции для нейронов скрытого слоя была выбрана сигмоидная функция:

$$F(x)=\frac{1}{1+e^{-\beta x}}, \tag{0.1}$$

а для нейронов выходного слоя – линейная. Здесь β – коэффициент, контролирующий «пологость» активационной функции.

2.1 Прямое распространение сигнала

Прямое распространение сигнала в нейросети описывается следующими формулами:

$$\begin{aligned} S^1_j &= \sum_{i=1}^{m} W^1_{ij}x_i + W^1_0, \\ u_j &= F(S^1_j),\ j=1,\ldots,n \end{aligned} \tag{0.2}$$

$$S_l^2 = \sum_{j=1}^{n} W_{jl}^2 u_j + W_0^2,$$
$$y_l = S_l^2, l = 1, \ldots, k \qquad (0.3)$$

Здесь x_i – внешний сигнал, поступающий на i-й нейрон входного слоя, u_j - сигнал, вырабатываемый j-м нейроном скрытого слоя, y_l - сигнал, вырабатываемый l-м выходным нейроном. Верхние индексы у весовых коэффициентов обозначают номер слоя. Отметим, что матрицы весов можно объединить с соответствующими матрицами смещений (W_0^1, W_0^2). В дальнейшем под матрицей весов будем понимать именно объединенную матрицу.

2.2 Настройка нейросети

Обучение нейронной сети происходит при помощи алгоритма обратного распространения ошибки, который, по сути, является методом градиентного спуска в пространстве весов. Назовем обучающей выборкой множество пар $\left(\vec{X}, \vec{D}\right)$, где $\vec{X} = \left(x_1, \ldots, x_m\right)^T$ - m-мерный входной вектор, $\vec{D} = \left(d_1, \ldots, d_k\right)^T$ k-мерный выходной вектор, такой, что если на вход нейросети подать $\vec{X}$, то она должна на выходе выработать вектор $\vec{D}$.

Процесс обучения состоит в следующем. В начальный момент всем весам нейросети с помощью датчика случайных чисел назначаются небольшие случайные значения. Входной вектор из каждого примера пропускается через нейросеть, в результате чего нейросетью вырабатывается результат *Y*, который сравнивается с желаемым *D*. В результате этого на каждом *p*-м примере вычисляется среднеквадратичная ошибка нейросети:

$$E_p = \frac{1}{2} \sum_{l=1}^{k} (d_l - y_l)^2, \qquad (0.4)$$

где *P* - общее число примеров, предъявляемых при обучении, а $y_к$ и d_k – соответственно компоненты векторов *Y* и *D*. Общая ошибка на всех примерах является суммой этих парциальных ошибок:

$$E = \sum_{p=1}^{P} E_p \qquad (0.5)$$

Процесс обучения состоит в минимизировании ошибки *E* относительно весовых множителей. В соответствии с методом градиентного спуска весовые коэффициенты на очередной итерации вычисляются по формуле:

$$W_{t+1} = W - \alpha \frac{\partial E}{\partial W}, \qquad (0.6)$$

где, α - параметр, характеризующий скорость сходимости.

Градиенты вычисляются по следующим формулам:

$$\frac{\partial E_p}{\partial W_{jl}^2} = (d_l - y_l) \frac{\partial F}{\partial S_l^2} u_j, \qquad (0.7)$$

$$\frac{\partial E_p}{\partial W_{ij}^1} = \left[\sum_{l=1}^{k} (d_l - y_l) \frac{\partial F}{\partial S_l^2} W_{jl}^2 \right] \frac{\partial F}{\partial S_j^1} x_i \qquad (0.8)$$

Признаком окончания обучения считается достижение некоторого порогового значения ошибки нейросети либо максимального номера эпохи.

3. Алгоритм обучения на GPU

Для того чтобы в полной мере использовать возможности параллельной системы, которой является видеокарта, алгоритм настройки нейросети должен быть соответствующим образом преобразован. Для уменьшения числа обменов с глобальной памятью видеокарты и увеличения отношения объема вычислений к объему загружаемых из нее данных по возможности все массивы данных должны быть записанными в двумерные массивы и обрабатываться на двумерной решетке. Для этого, например, всю обучающую выборку, т.е. совокупность пар $\left(\vec{X}, \vec{D}\right)$, поместим в две матрицы с размерами $P \times M$ и $P \times K$ соответственно. С этой же целью избавляемся от матриц, хранящих ошибки обучения в текущей эпохе, заменив их одним массивом для хранения весов $W_{N \times K}^2$, рассчитанных на предыдущей операции.

Естественно, что если число обучающих примеров очень велико, то эти матрицы придется разбивать на блоки. В этом случае пакетная обработка обучающей выборки позволяет снизить частоту обменов между CPU и GPU, которая приводит к существенному замедлению всего алгоритма. В результате этого при прямом распространении сигнала операция умножения матрицы весов на вектор, получаемый с входа нейронов, заменятся на операцию перемножения двух матриц:

$$Y_{P \times K} = F(X_{P \times M} W_{M \times N}^1) W_{N \times K}^2 \qquad (0.9)$$

Наконец, изменив порядок вычислений, выделим ядра потоковой обработки, выполняющиеся на видеокарте, по возможности преобразуя схему вычислений так, чтобы во время обработки данных оперировать блоками нужного размера и обеспечить загруженность процессорных элементов, способную скрыть задержку при доступе к глобально памяти видеокарты. Это также позволит ускорить обработку за счет такой особенности параллелизма видеокарты, как векторность ПЭ графической карты.

В предлагаемом алгоритме настройка нейросети выполняется в цикле обучения последовательно следующими ядрами (0.10), (0.11), (0.12), (0.13), (0.14):

Ядро GPU_PropagateW1 реализует вычисление сигнала, поступающего на скрытый слой:

$$U_{P \times N} = F(X_{P \times M} W_{M \times N}^1) \qquad (0.15)$$

Ядро GPU_PropagateW2 реализует вычисление выходного сигнала сети:

$$Y_{P \times K} = U_{P \times N} W_{N \times K}^2 \qquad (0.16)$$

Ядро GPU_BackpropW2 реализует вычисление каждого элемента матрицы $W_{N \times K}^2$ на новом шаге настройки сети по формуле:

$$W_{jk}^2 = W_{jk}^2 - \alpha \sum_{p=1}^{P} (Y_{pk} - D_{pk}) U_{pj} \qquad (0.17)$$

Ядро GPU_BackpropW1AccumHid вычисляет элементы вспомогательной матрицы $U'_{P \times N}$, такой, что:

$$U'_{pj} = \sum_{k=1}^{K} (Y_{pk} - D_{pk}) W_{jk}^{2} \quad (0.18)$$

Ядро GPU_BackpropW1 вычисляет элементы матрицы $W_{M \times N}^{1}$ на новом шаге обучения:

$$W_{ij}^{1} = W_{ij}^{1} - \beta\alpha \sum_{p=1}^{P} \left[U_{pj}(1 - U_{pj}) X_{pi} U'_{pj} \right] \quad (0.19)$$

Псевдокод алгоритма обучения нейросети приведен на Рис. 3.

```
Iter = 0;

X_Matrix = learnDataSet_InputVectors
D_Matrix = learnDataSet_OutputVectors

Do
      // прямое распространение
      U_Matrix = GPU_PropagateW1()
      Y_Matrix = GPU_PropagateW2()

      PREV_W2_Matrix = W2_Matrix;

      // обратное распространение для весов W2
      GPU_BackpropW2()

      // обратное распространение для весов W1
      GPU_BackpropW1AccumHid()
      GPU_BackpropW1()

      Iter = iter + 1
While (iter < maxIter)
```

Рис. 3. Псевдокод алгоритма обучения нейросети.

Очевидно, что введение в схему вычислений матрицы разности $deltaYD_{PxK} = Y_{PxK} - D_{PxK}$, вычисляемой ядром GPU_PropagateW2, уменьшит суммарное количество загрузок из глобальной памяти и количество операций.

Что касается вычисления ошибки нейросети – это типичная операция параллельной редукции массива данных. Особенности реализации этой операции на GPU, в том числе и GPU с технологией CUDA, описаны, например, в работах [9,10,11]. В нашем случае матрица $deltaYD_{PxK} = Y_{PxK} - D_{PxK}$ редуцируется по формуле:

$$Err_{iter} = \frac{1}{2} \sum_{i=1}^{PK} deltaYD_{i}^{2} \quad (0.20)$$

Этот шаг в виде отдельного ядра может быть без труда интегрирован в поток обработки и в данной работе не рассматривается.

4. Реализация на GPU

Для написания прототипа программы, реализующей алгоритм с использованием GPU, из соображений скорости разработки была выбрана высокоуровневая библиотека для программирования многоядерных систем RapidMind v2.0 [12]. На ней исследовалась возможность ускорения обучения нейросети традиционными методами GPGPU – через графический интерфейс видеокарты. Результаты были использованы при разработке модуля обучения при помощи технологии CUDA.

Архитектура CUDA построена вокруг масштабируемого массива SIMD-мультипроцессоров, каждый из которых состоит из восьми скалярных процессоров, обладающих наборами 32-битных регистров. Каждый мультипроцессор обладает разделяемой кэш-памятью, доступной для чтения и записи, текстурным кэшем и кэшем констант, данные в которых доступны только для чтения.

При запуске ядра потоковой обработки вычисляемая сетка разбивается на блоки SIMD-нитей. Нити внутри одного блока имеют возможность обмениваться данными посредством общей разделяемой памяти и могут быть синхронизированы операцией типа барьер. Отдельные блоки могут общаться только посредством глобальной памяти и не могут быть синхронизированы. Способ разбиения сетки указывается программистом и для достижения оптимальной занятости мультипроцессоров должен быть тщательно подобран с учетом количества используемых регистров и объема необходимой ядру кэш памяти [13].

Вернемся к описанию реализации алгоритма на GPU. Все матрицы данных, участвующие в вычислениях, помещаются в память видеокарты: $X_{P\times M}$, $D_{P\times K}$, $deltaYD_{PxK}$, $U_{P\times N}$, $U'_{P\times N}$, $W^1_{M\times N}$, $W^2_{N\times K}$ и $\bar{W}^2_{N\times K}$ - массив весов, вычисленных на предыдущей итерации. Формат матриц в памяти – по строкам (row-major). При обработке данные, вычисляемые ядрами, разбиваются на блоки $16\times16, 64\times16, 16\times1$.

Нетрудно заметить, что ядра GPU_PropagateW1 и GPU_PropagateW2,– по сути программы перемножения матриц. Существует большое количество работ, посвященных реализации различных операций линейной алгебры на GPU [5,6,7]. В данной работе использовался блочный алгоритм с размерами блоков, подобранными согласно [8]. При этом каждый перемножаемый массив делился на блоки одного из размеров $16\times16, 64\times16, 16\times1$.

5. Результаты тестирования

Для проверки эффективности предложенного метода ускорения процесса настройки нейросети использовалась компьютер со следующими параметрами:

Таблица 1. Аппаратная конфигурация системы тестирования.

CPU:	Intel Core 2 Duo 3.00 GHz
RAM:	3.25 Gb
GPU:	GeForce 8800 GT, 256 RAM

Расчеты проводились при различном числе нейронов скрытого слоя. Алгоритм для CPU был реализован на ansi C с использованием одного их двух доступных ядер процессора. Каждый из тестируемых модулей загружал данные из заранее сгенерированного набора. Обучение продолжалось фиксированное число итераций, после чего сравнивалось среднее время одной итерации обучения на GPU и CPU. На Рис. 4 изображены графики ускорения в зависимости от числа нейронов скрытого слоя при фиксированных на значениях 512 и 256 трех оставшихся параметрах: количестве векторов в обучающей выборке, нейронов входного и выходного слоев.

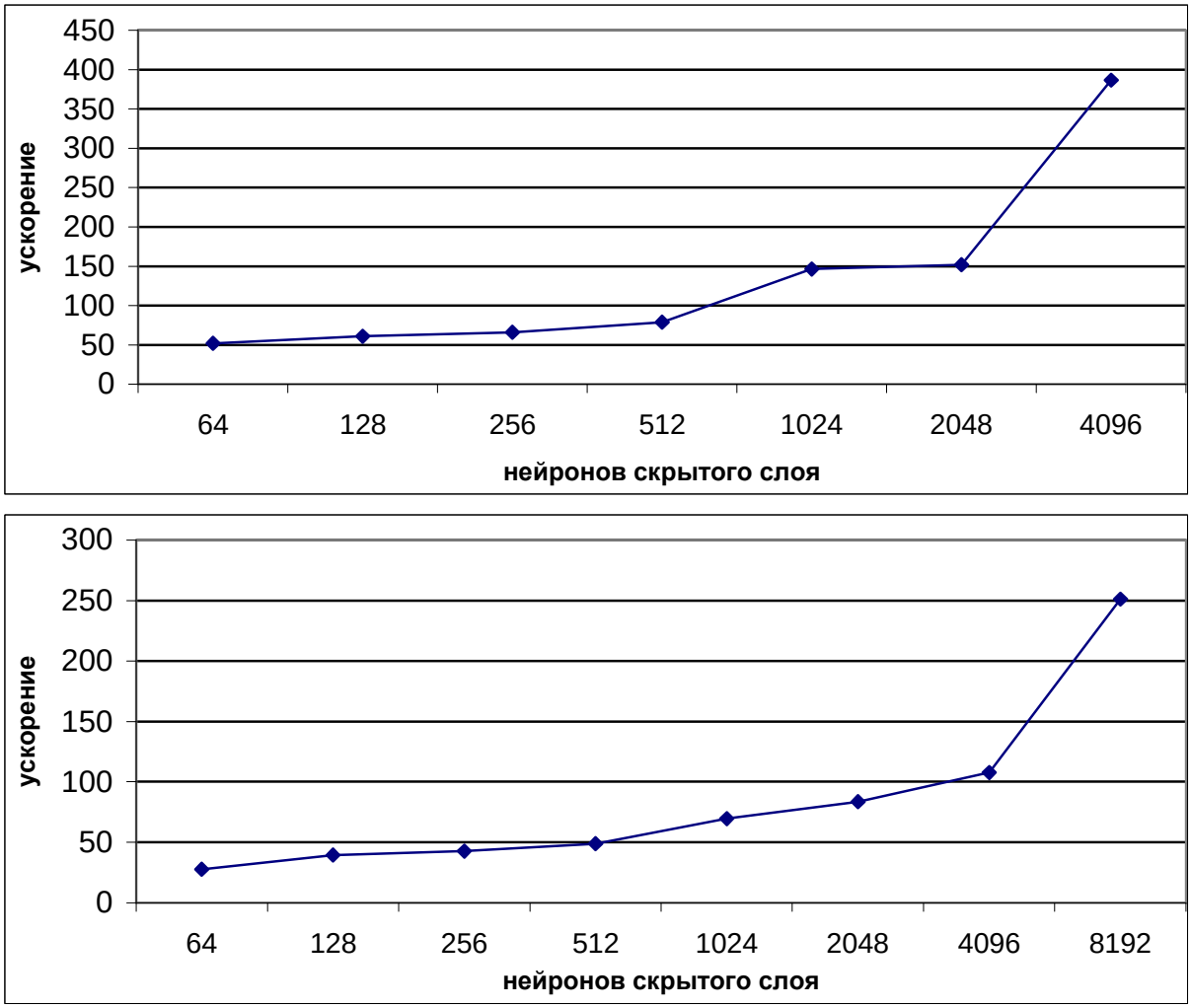

Рис. 4. Коэффициент ускорения процесса настройки нейросети в зависимости от числа используемых нейронов. На графике сверху количестве векторов в обучающей выборке, нейронов входного и выходного слоев фиксированы на значении 512, снизу – на 256.

На графиках видно, как с увеличением размера обрабатываемых данных ускорение растет, достигая нескольких порядков.

6. Заключение

В данной работе был предложен метод реализации обучения трехслойной нейросети на графической карте. Результаты тестирования нашего алгоритма показали значительное ускорение относительно традиционной версии на CPU, делая возможным обучение нейросети в интерактивном режиме, например, в задачах распознавания образов, прогнозирования временных рядов, аппроксимации экспериментальных данных и других.

Для улучшения представленных результатов необходимо провести дополнительную оптимизацию потоковых ядер и подобрать оптимальные размеры блоков аппаратных нитей. Возможная модификация модельной нейросети, например, изменение вида активационной функции или увеличение количества скрытых слоев, не должна внести принципиальных изменений в схему вычислений, таким образом, стоит ожидать, что порядок величины ускорения обучения модифицированной сети сохранится.

Мы убеждены, что современные графические карты способны внести свой вклад в высокопроизводительные вычисления, а количество проектов, использующих параллелизм видеокарты для вычислений общего назначения, будет увеличиваться.

Литература

1. S. Haykin. Neural Networks: A Comprehensive Foundation (2nd Edition). -Prentice Hall, 1998. -p.842
2. History of GPGPU:

   [http://www.gpgpu.org/data/history.shtml], сентябрь 2008
3. John D.Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. // Eurographics 2005, State of the Art Reports. -August 2005. -pp. 21-51.
4. Mark Harris. Mapping Computational Concepts to GPUs. // GPU Gems 2. -Addison Wesley, December 2006. -pp.493-508
5. J. Krüger and R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. // Proceedings of SIGGRAPH 2003. –July 2003. -vol. 22, no. 3. -pp. 908-916
6. J. Krüger and R. Westermann. A GPU Framework for Solving Systems of Linear Equations. // GPU Gems 2. -Addison Wesley, December 2006. -pp.703-718
7. J. D. Hall, N. A. Carr, and J. C. Hart. Cache and Bandwidth Aware Matrix Multiplication on the GPU. Technical Report. -UIUCDCS-R-2003-2328, 2003:

   [http://graphics.cs.uiuc.edu/~jch/papers/UIUCDCS-R-2003-2328.pdf], сентябрь 2008
8. Vasily Volkov, James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra:

   [http://parlab.eecs.berkeley.edu/pubs/volkov-benchmarking.pdf]
9. Daniel Horn. Stream Reduction Operations for GPGPU Applications. // GPU Gems 2. -Addison Wesley, December 2006. -pp. 573-589
10. Mark Harris. Optimizing Parallel Reducion in CUDA. NVidia Technical Report:

    [http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf], январь 2009
11. M. Harris, S. Sengupta, J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. // GPU Gems 3. -Addison Wesley, December 2007. -pp. 851-875
12. The RapidMind Multi-Core Development Platform:

    [http://www.rapidmind.com/pdfs/WP_RapidMindPlatform.pdf], февраль 2009
13. CUDA 2.0 Programming Guide:

    [http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf], сентябрь 2008