6 КОМПЬЮТЕРНЫЕ СИСТЕМЫ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

УДК 004.272:004.032.26

РЕАЛИЗАЦИЯ РАДИАЛЬНО-БАЗИСНОЙ НЕЙРОННОЙ СЕТИ НА МАССИВНО-ПАРАЛЛЕЛЬНОЙ АРХИТЕКТУРЕ ГРАФИЧЕСКОГО ПРОЦЕССОРА

Н.О. Матвеева

Предлагается распараллеливание в технологии программно-аппаратной архитектуры (CUDA) алгоритма обучения радиально-базисной нейронной сети (RBFNN), основанного на идее последовательной настройки центров, ширины и весов сети, а также идее коррекции весов по алгоритму минимизации квадратичного функционала методом сопряженных градиентов. Приводятся результаты сравнения времени обучения RBFNN на различных центральных и графических процессорах, доказывающие эффективность распараллеливания.

Ключевые слова: графический процессор, параллельность, CUDA, массивно-параллельная архитектура, RBFNN, дифференциальное уравнение.

Введение

Для решения краевых задач математической физики, описываемых дифференциальными уравнениями в частных производных, наибольшее распространение получили методы конечных разностей и конечных элементов, но эти методы требуют построения сеток и позволяют получить решение только в узлах сетки. Этого существенного недостатка лишены бессеточные методы [1]. Бессеточные методы эффективно реализуются на RBFNN [2]. Главное достоинство RBFNN состоит в использовании принципов обучения для формирования оптимальных параметров радиально-базисных функций (RBF). Такие методы могут быть эффективно реализованы на параллельных системах.

Для распараллеливания RBFNN наиболее подходит модель вычислений SIMD (Single Instruction – Multiple Data), так как нейроны одного слоя сети выполняют одинаковые действия над различными данными. На основе такой модели вычислений построены современные графические процессоры (GPU). GPU выигрывают у кластерных систем по критерию цена/производительность.

Целью работы является разработка параллельной реализации алгоритма обучения RBFNN для решения краевых задач математической физики на графическом процессоре с использованием технологии CUDA и исследование эффективности распараллеливания путем сравнения времени решения краевой задачи на GPU и на центральном процессоре (CPU).

Структура RBFNN

RBFNN (рис. 1) представляет собой сеть с двумя слоями. Первый слой осуществляет преобразование входного вектора $\mathbf{x}$ с использованием RBF. Практически используются различные RBF.

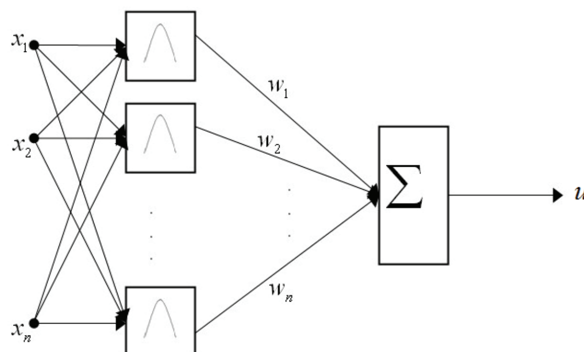

Рис. 1. Радиально-базисная нейронная сеть

В дальнейшем будем использовать наиболее часто употребляемую функцию – гауссиан, имеющий для k-го нейрона вид

$$\varphi_k(\mathbf{x}) = \exp(-r_k^2 / a_k^2), \quad (1)$$

где $\mathbf{x}$ – входной вектор; $r_k = \|\mathbf{x} - \mathbf{c}_k\|$ – радиус k-го нейрона; a_k – ширина k-го нейрона; $\mathbf{c}_k$ – центр k-го нейрона. Выход сети описывается выражением

$$u = \sum_{k=1}^{m} w_k \varphi_k(\mathbf{x}), \tag{2}$$

где w_k – вес связи выходного нейрона с k-ым нейроном первого слоя; m – число нейронов первого слоя.

Алгоритм обучения RBFNN

Рассмотрим градиентный алгоритм обучения RBFNN на примере решения двумерного уравнения Пуассона

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y),\ (x, y) \in \Omega, \tag{3}$$

$$u = p(x, y),\ (x, y) \in \partial\Omega, \tag{4}$$

где $\partial\Omega$ – граница области; f и p – известные функции (x, y). Выбирая в качестве RBF гауссиан (1), определяемый как $\varphi_k(x, y) = \exp\left(-\frac{r_k^2}{a_k^2}\right)$, рассмотрим RBF-сеть как аппроксиматор функции решения

$$u(\mathbf{x}) = \sum_{k=1}^{m} w_k \varphi_k(r_k), \tag{5}$$

где m – число RBF (скрытых нейронов); $r_k = \sqrt{(x - c_{xk})^2 + (y - c_{yk})^2}$ **;** (c_{xk}, c_{yk}) **–** координаты центра нейрона k **;** a_k **–** ширина нейрона k.

Обучение сети сводится к настройке весов, расположения центров и ширины нейронов, минимизирующих функционал качества (функционал ошибки), представляющий собой сумму квадратов невязок в контрольных точках

$$I(\mathbf{w}, \mathbf{c}, \mathbf{a}) = \frac{1}{2}\sum_{i=1}^{N}\left[\frac{\partial^2 u(x_i, y_i)}{\partial x^2} + \frac{\partial^2 u(x_i, y_i)}{\partial y^2} - f(x_i, y_i)\right]^2 + \frac{\lambda}{2}\sum_{j=1}^{K}\left[u(x_j, y_j) - p_j\right]^2, \tag{6}$$

где λ – штрафной множитель; p_j – значение граничных условий первого рода в точке j-границы; N и K – количество внутренних и граничных контрольных точек.

В [2] предложен алгоритм последовательной настройки центров, ширины и весов с использованием метода градиентного спуска с подбором коэффициентов скорости обучения. Подбор коэффициентов скорости обучения является основным недостатком данного метода.

В [3] для настройки весов предложен другой алгоритм, который сформулирован как задача минимизации методом сопряженных градиентов квадратичного функционала. Данный метод не содержит подбираемых коэффициентов, и настройка весов происходит быстрее, чем методом градиентного спуска.

Очень важно соблюдать при обучении соотношение между оптимальным количеством нейронов m и количеством контрольных точек [4]

$$m \sim \sqrt[3]{N + K},$$

где ~ – знак пропорциональности. Данное соотношение требует большого числа контрольных точек, что ведет к увеличению времени решения задачи. В [3] обосновано, что многократная случайная генерация относительно небольшого числа контрольных точек внутри и на границе области решения компенсирует нарушение пропорции.

Таким образом, обучение радиально-базисной сети состоит из следующих основных шагов.

1. Генерация начальных значений параметров сети.
2. Генерация случайных контрольных точек для одного или нескольких циклов обучения.
3. Несколько циклов настройки центров и ширины при зафиксированных весах. В данном случае центры и ширина настраивались методом градиентного спуска:

$$c_k^{(n)} = c_k^{(n-1)} - \beta^{(n-1)} \frac{\partial I(c_k^{(n-1)}, a_k^{(n-1)}, w_k^{(n)})}{\partial c_k^{(n-1)}},$$

$$a_k^{(n)} = a_k^{(n-1)} - \alpha^{(n-1)} \frac{\partial I(c_k^{(n-1)}, a_k^{(n-1)}, w_k^{(n)})}{\partial a_k^{(n-1)}}.$$

4. Несколько циклов настройки весов при зафиксированных значениях центров и ширины по алгоритму минимизации методом сопряженных градиентов квадратичного функционала, полученного в [3]: $I = (\mathbf{Aw}, \mathbf{w}) - 2(\mathbf{s}, \mathbf{w}) + 0{,}5(\mathbf{f}, \mathbf{f}) + 0{,}5\lambda(\mathbf{p}, \mathbf{p})$,
   где $\mathbf{A} = \frac{1}{2}(\mathbf{M}^T\mathbf{M} + \lambda\mathbf{N}^T\mathbf{N})$ – симметричная положительно определенная матрица; $\mathbf{s} = \frac{1}{2}(\mathbf{M}^T\mathbf{f} + \lambda\mathbf{N}^T\mathbf{p})$; $\mathbf{M}$ – матрица $N \times m$ с элементами $m_{ik} = 4\exp\left(-\frac{r_{ik}^2}{a_k^2}\right)\frac{r_{ik}^2 - a_k^2}{a_k^4}$; $\mathbf{N}$ – матрица $K \times m$ с элементами $n_{ik} = \exp\left(-\frac{r_{ik}^2}{a_k^2}\right)$; $\mathbf{f}$ – вектор значений функции правой части во внутренних точках; $\mathbf{p}$ – вектор граничных условий в граничных контрольных точках.
5. Проверка, достигнута ли требуемая погрешность. Для этого целесообразно использовать одну из норм вектора невязки (вектор невязки вычисляется во внутренних и граничных контрольных точках). Если требуемая погрешность не достигнута, то переход на шаг 2.

Используется пакетный режим обучения, т.е. вычисляется усредненная ошибка по всем контрольным точкам.

Алгоритмы распараллеливания вычислений на графическом процессоре в технологии CUDA

Как было описано выше, для минимизации весов сети использовался алгоритм минимизации квадратичного функционала методом сопряженных градиентов. Непосредственно данный алгоритм состоит из матрично-векторных операций, распараллеливание которых описано в [5]. Но, кроме коррекции весов, большая доля времени приходится на вычисление матриц $\mathbf{M}$ и $\mathbf{N}$, расстояний r_{ik}, коррекцию векторов центров и ширины. При этом элементы данных матриц и векторов находятся не с помощью матрично-векторных операций, а в циклах, что делает невозможным использование стандартной библиотеки CUDA CUBLAS. Например, вычисление матрицы $\mathbf{M}$ и вектора $\mathbf{r}_1$ осуществляется в двух вложенных циклах (рис. 2). Реализация данной функции на графическом процессоре позволяет избавиться от обоих циклов и параллельно вычислить каждый из элементов матрицы **M**.

Функция (входные данные: вектора – **w**, **a**, **f**; матрица – **R**; числа – *m*, *n*ν)
$\mathbf{r}_1$ = нулевой вектор 1×*n*ν;
M = нулевая матрица *n*ν×*m*
Цикл 1: *i* от 1 до *n*ν
Цикл 2: *k* от 1 до *m*
$$m_{ik} = 4e^{-\frac{r_{ik}^2}{a_k^2}}\frac{r_{ik}^2 - a_k^2}{a_k^4}$$
$$r_{1i} = r_{1i} + m_{ik}w_k$$
Конец цикла 2
$$r_{1i} = r_{1i} + f_i$$
Конец цикла 1
Возвращаемые значения: M, $\mathbf{r}_1$

Рис. 2. Алгоритм последовательного вычисления матрицы **M** и вектора $\mathbf{r}_1$

Для вычисления одного элемента матрицы $\mathbf{M}$ требуется прочитать из памяти r_{ik} и a_k и выполнить все операции по формуле $m_{ik} = 4\exp\left(-\frac{r_{ik}^2}{a_k^2}\right)\frac{r_{ik}^2 - a_k^2}{a_k^4}$, т.е. необходимо небольшое количество чтений из памяти и большое количество вычислений. Такие задачи прекрасно реализуются на архитектуре графического процессора, так как слабое место GPU – относительно медленное чтение из памяти – компенсируется за счет большого количества параллельно выполняемых вычислений.

Как видно из рис. 3, вычисления на графическом процессоре происходят в *n*ν блоках, каждый из которых содержит *m* потоков. Один поток вычисляет один элемент матрицы $\mathbf{M}$, в результате все эле-

менты матрицы вычисляются параллельно. Вычисление вектора $r_{1i} = \sum_{k=1}^{m} m_{ik} w_k - f_i$ реализуется в этом том же ядре, что и матрицы $\mathbf{M}$, для уменьшения чтений из памяти, так как найденные элементы матрицы $\mathbf{M}$ сразу используются для вычисления вектора $\mathbf{r}_1$ через разделяемую память блока размером $1 \times m$.

Рис. 3. Алгоритм параллельного вычисления матрицы **M** и вектора $\mathbf{r}_1$ на GPU

Другие вычисления алгоритма обучения RBFNN, такие как вычисление матрицы $\mathbf{N}$, коррекция центров и ширины методом градиентного спуска, метод сопряженных градиентов для настройки весов, распараллеливаются на графическом процессоре подобным образом.

Анализ результатов экспериментов

Алгоритм обучения RBFNN реализовывался с помощью технологии CUDA на графическом процессоре nVidia GeForce 8800 GTX, на центральных процессорах Intel Pentium 4 3 GHz (CPU1) и Intel® Core™ 2 Quad CPU Q8200 2,33GHz (CPU2) на Microsoft Visual Studio 2008. Экспериментальное исследование проводилось на примере модельной задачи (3)–(4) для $f(x, y) = \sin(\pi x)\sin(\pi y)$, $p(x, y) = 0$. Анализ результатов экспериментов показал, что часть алгоритма, не включающая обучение весов сети, распараллеливается очень эффективно, значительно уменьшая время вычислений. Метод сопряженных градиентов для обучения весов становится более эффективным при увеличении числа нейронов. Для решаемой задачи увеличивать количество нейронов не имело смысла, поэтому ускорения были получены в основном за счет распараллеливания шага 1, вычисления матриц $\mathbf{M}$, $\mathbf{N}$ и др., что можно видеть из таблицы. При решении более сложного уравнения эффективность распараллеливания на графическом процессоре будет значительней, но и для решаемой задачи было получено существенное ускорение. Общее ускорение вычислялось по формуле $S = \frac{\text{Время вычислений CPU}}{\text{Время вычислений GPU}}$. В сравнении с CPU Intel Pentium 4 3 GHz при числе контрольных точек 224 оно составило 38,58, при числе контрольных точек 524 – 83,42, в сравнении с Intel® Core™ 2 Quad CPU Q8200 2,33GHz – 17,31 и 34,5 соответственно.

В таблице приведены результаты экспериментов, из которых видно, что уже при небольшом числе нейронов и контрольных точек реализация алгоритма обучения на графическом процессоре дает существенный выигрыш во времени.

Параметры сети	Шаг алгоритма обучения сети	GPU, мс	CPU 1, мс	CPU 2, мс	Уск. 1	Уск. 2
Число контр. точек = 224	Шаг 3	0,56	64,1	27,8	114,5	49,6
Число контр. точек = 524	Шаг 3	0,58	173,2	71,7	298,6	123,6
Число нейронов = 64	Шаг 4	1,12	1,9	1,2	1,7	1,07

Таблица. Результаты экспериментов

Заключение

В результате распараллеливания алгоритма обучения RBFNN, основанного на идее последовательной настройки центров, ширины и весов и идее использования для настройки весов алгоритма минимизации квадратичного функционала методом сопряженных градиентов, было достигнуто существенное уменьшение времени вычислений. При решении уравнения Пуассона, с использованием сети с 64-мя нейронами и 524 контрольными точками, достигнуто ускорение в 34,5 раза в сравнении с CPU Intel® Core™ 2 Quad.

Работа выполнена по тематическому плану научно-исследовательских работ Пензенского государственного педагогического университета, проводимых по заданию Федерального агентства по образованию.

Литература

1. Liu G.R. An Introduction to Meshfree Methods and Their Programming / G.R. Liu, Y.T. Gu. – Springer, 2005. – 479 p.
2. Jianyu L. Numerical solution of elliptic partial differential equation using radial basis function neural networks / L. Jianyu, L. Siwei, Q. Yingjiana, H. Yapinga // Neural Networks. – 2003. – 16(5/6). – P. 729–734.
3. Горбаченко В.И. Радиально-базисные нейронные сети для решения краевых задач бессеточными методами / В.И. Горбаченко, Е.В. Артюхина, В.В. Артюхин // Нейроинформатика-2010: Сборник научных трудов XII Всероссийской научно-технической конференции. В 2-х частях. Часть 2. – М.: НИЯУ МИФИ, 2010. – С. 237–247.
4. Хайкин С. Нейронные сети: Полный курс / С. Хайкин. – М.: Вильямс, 2006. – 1104 с.
5. Горбаченко В.И. Реализация итерационных алгоритмов решения систем линейных алгебраических уравнений на графических процессорах в технологии CUDA / В.И. Горбаченко, Н.О. Матвеева // Вопросы радиоэлектроники. Серия ЭВТ. – 2008. – Вып. 6. – С. 65–75.

Матвеева Наталья Олеговна – Пензенский государственный педагогический университет им. В. Г. Белинского, аспирант, fire_tiger_705@mail.ru