МЕТОДЫ ОПТИМИЗАЦИИ ПРОГРАММ ЭМУЛИРУЮЩИХ НЕЙРОННЫЕ СЕТИ НА INTEL-СОВМЕСТИМЫХ ПРОЦЕССОРАХ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ SIMD (SINGLE INSTRUCTION MULTIPLE DATA)

Скрибцов П., skribtsov@physicon.ru,

Научный Центр Нейрокомпьютеров
Новая Басманная-20, Москва 107066, scn@mail.cnt.ru

1. Введение

В настоящее время аппаратные ускорители нейросетевых алгоритмов еще не получили массовое распространение, например, по сравнению по распространнености плат аппаратного ускорение машинной графики. Однако изучение нейронных сетей возможно и без использование аппаратных средств, при помощи программ, эмулирующих работу нейронной сети на однопроцессорной машине. В этом случае возникает проблема нехватки скорости обработки данных. Данная статья посвящается изучению методов оптимизации программ, эмулирующих нейронные сети, с использованием знаний об архитектурных особенностях процессоров Intel, позволяющих добиться существенного выйгрыша в производительности.

2. Математическое определение задачи

В данной короткой статьне невозможно описать оптимизацию для всех типов нейронных сетей. Однако для большинства типов нейронных сетей требуются вычислять достаточно простые математические операции. Результаты, полученные в данной статье легко обобщить на нейронную сеть произвольного типа. Объектом оптимизации в нашем случае будет полносвязанная последовательная нейронная сеть. Каждый слой производит следующую операцию:

$$y_j = f(\sum_i x_i w_{ij} + w_{0j}) \qquad (1)$$

где y_j - компонента выходного вектора,

x_i - компонента входного вектора

w_{ij} - матрица весов слоя

w_{0j} - «свободные веса»,

f - сигмоидальная функция активации, в нашем случае предположим $f(x) = \frac{2}{\pi}\arctan(x)$

Наша задача будет проанализировать типичную структуру нейронной сети и предложить ряд рекоммендаций которые позволят улучшить код и увеличить производительность. Однако перед тем как приступать к низкоуровневой оптимизации, с использованием знаний об архитектурных особенностях процессоров семейства Intel необходимо произвести предварительную оптимизацию кода, включающую устранение процессов выделения памяти, вызова вирутальных функций, итп.

3. Оптимизация представления данных и кода

Предварительную оптимизацию можно считать завершенной когда отладчик (профилировщик) указывает на то, что программа во время основного цикла вычислений 99% времени проводит в отрывке кода похожего на нечто подобное:

```
float sum = 0;
for(int j = 0; j< inputs.length; j++) {
        sum += weights[neuron][j]*input[j]
}
output[neuron] = constant*Math.atan(sum);
```

что есть ни что иное как программное представление формулы (1).

С этого момента (по завершению предварительной оптимизации) производительность программы всецело зависит от эффективности компилятора. В настоящее время существует достаточно большое количество так называемых «оптимизирующих» компиляторов, которые генерируют более быстрый код. Один из самых популярных «оптимизирующих» компиляторов это **Intel C++ compiler**, но , к сожалению, он не является бесплатным. Использование «оптимизирующего» компилятора, это хороший шаг к увеличению производительности программмы, но к сожалению этот шаг не лишен своих минусов:

а) оптимизирующий компилятор не всегда полностью «понимает» специфику алгоритма и поэтому не всегда может использовать ее для оптимизации машинного кода.

b) Достаточно трудно понять, что на самом деле делает компилятор с программой, представленной в машинном коде, поэтому не может быть уверенности в 100%-ой оптимальности решения, особенно учитывя наличие у компилятора различных параметров, от которых зависит как будет сгенерирован код
c) если в компиляторе есть ошибка, то исправить или «обойти» ее самостоятельно будет невозможно, кроме , возможно, методом отключения «сомнительных» параметров компилятора.

Даже если написание программы на языке низкого уровня (ассмеблере) не входт в ваши планы, следующая информация может оказаться полезной для представления о том, каким образом производятся оптимизации, что возможно облегчит насройку вашего компилятора.

Чтобы понять как производится оптимизация кода и представления данных (что как показывает практика немаловажно) необходимо представлять себе основы архитектуры современного процессора. (см. Рис 1).

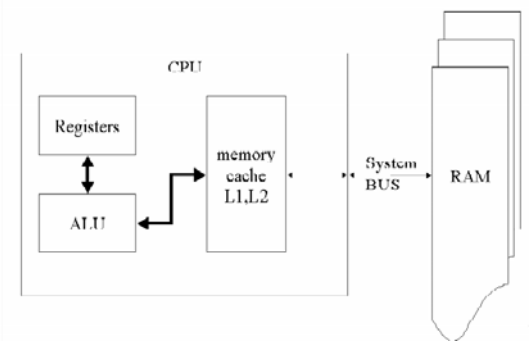

Рис 1.

Основная проблема с данными заключается в том, что арифметическо-логическое устройство (ALU – arithmetical-logical unit) может производить быстрый обмен данными с регистрами и кэшем (внутренним ОЗУ процессора) в то время как обмен данными с внешней памятью медленнее на порядок величины. Когда программа обращается к какому либо адресу, если этот участок памяти еще не «отображен» в кэше, тогда чтение и запись происходит со скоростью работы системной шины, что достаточно медленно по сравнению с тактовой частотой самого процессора. Как только к данным обратились, если не указано иначе, содержимое участка памяти копируется в кэш. Если программа повторно обращается к этому участку памяти, то обмен происходит со скоростью работы тактового генератора процессора. Поскольку размер кэша ограничен, процессор может удерживать в кэше только определенное количество участков памяти. Из этого можно сделать несколько выводов:

a) необходимо по максимальной возможности использовать регистры процессора для хранения значений переменных вместо оперативной памяти, это особенно относится к счетчикам циклов и других часто используемых переменных. Стоит отметить, однако, что обмен данными между регистрами MMX и регистрами общего назначения достаточно медленное, к примеру, комманда,
movd eax, mm0
занимает целых несколько тактов
b) хранить все переменные использующиеся в теле главного цикла близко друг к другу.
c) повторно использовать те же участки памяти для различных (всех) целей, если возможно. К примеру тот же самый участок памяти может использоваться для хранения входных и выходных векторов слоя нейронной сети, как показано на Рис. 2.

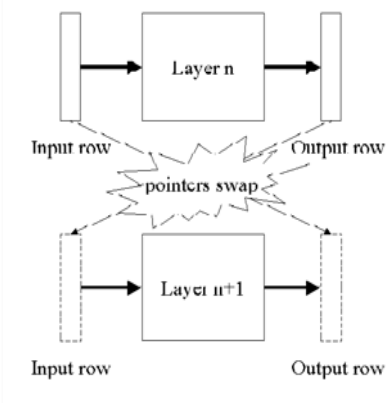

Рис 2.

d) Использовать стэк для хранения промежуточных данных. Поскольку область памяти выделенная под стек используется довольно часто, высока вероятность того, что этот участок памяти будет уже прокеширован. Стэк может быть эффективно использоваться в алгоритме обучения нейронной сети для хранения промежуточной информации, собранной во время прямого прохода (например выходные значчения нейронов, итп.), поскольку эта информация потребуется во время обратного прохода в «обратной последовательности». Структура данных в данном случае хорошо «ложится» на поведение стэка.
e) необходимо избегать частых смен «чтение\запись» в память.

Это базовые знания и базовые выводы об оптимизации структуры данных, чтобы сделать более глубокие выводы необходимо ознакомиться более подробно с архитектурой конкретного процессора (или например семейства процессоров IA-32) под которые пишется программа.

Особый принцип, называемый SIMD (Single Instruction Multiple Data) реализованный в расширенных наборах комманд современных процессоров под именами SSE (Streaming SIMD Extension), 3DNow!, SSE2 крайне удобен для оптимизации кода производящего нейровычисления. Основной принцип состоит в том, что одна комманда выполняет некоторую элементарную математическую операцию сразу над несколькими парами операндов, при этом эта комманда выполняется за существенно меньшее число тактов, чем занял бы аналогичный код для стандартного математического сопроцессора (FPU). SSE расширение доступно во всех процессорах совместимых с Pentium-III, включая также процессор Athlon XP фирмы AMD.

К примеру такая операция как

addps xmm0, xmm1

выполняет следующие действия над специальными 128-битовыми регистрами (XMM registers), которые могут хранить 4 числа с плавающей точкой одинарной точности (см Рис. 3)

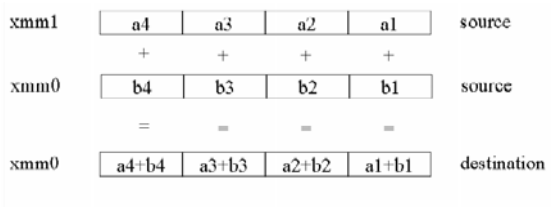

Рис 3.

И данная комманда занимает всего лишь 2 такта! Стоит также отметить что во время выполнения комманд SSE сопроцессор не используется, что позволяет параллельно запускать вычисления в FPU.

Можно сделать краткие выводы:

a) Необходимо корректировать структуру архитектуры вашей нейронной сети для того, чтобы код было легче оптимизировать с использованием SIMD. Например, так как SSE инструкции работают с 4 операндами, значит имеет смысл делать количество нейронов в слое кратным 4 (для больших сетей это несущественное ограничение, т.к. нейронные сети с большим числом нейронов ведут себя схоже при небольшом изменении числа нейронов).
b) Необходимо корректировать код программы, чтобы он эффективно использовал особенности конкретной нейронной сети. Например, если нейронная сеть имеет 3 входа (например нейронная сеть для обработки 3D данных), то код для обработки первого слоя может быть отличным от кода для обработки других слоев, это может также потребовать хранить веса для первого слоя «по столбцам», а не «строчкам».

Вместо самостоятельного написания ассемблерного кода с использованием SIMD можно воспользоваться готовой библиотекой, такой как например Intel® Signal Processing Library 4.5. Однако практика показывает, что специализированный код «подогнанный» очевидно работает эффективней.

4. Выводы

Техника, описанная выше была протестирована на процессоре AMD Athlon 1.3GHz с 133-мегагерцовой шиной. Интерфейс был написан на языке Java с вызовами к native methods. DLLки для native methods были написаны на C++ с ассемблерными вставками. Для компиляции SSE инструкций использовалась Microsoft Visual Studio .NET. Средняя производительность программы после всех шагов оптимизации для данной машины привысила 2Giga flops/sec. Что дало выйгрыш в скорости обучения нейронной сети по сравнению с неоптимизированным кодом на языке Java приблизительно в 40 раз.

Литература

[1] Neural Networks Theory, Galushkin A. I. Radiotechnika,. Moscow 2000
[2] Assembler Special Reference, Victor Yurov, Sankt-Petersburg, 2001, #ISBN 5-272-00119-2
[3] Processors Pentium-4, Athlon and Duron, Michael Guk, Victor Yurov, Sankt-Petersburg, 2001, #ISBN 5-318-00559-4
[4] Intel Performance Library http://developer.intel.com/software/products/perflib/
[5] Example of real time ray-tracing implementation using SSE optimization, Lev Dymchenko, http://www.virtualray.ru/eng/engine.htm

APPLICATION OF SIMD TECHNOLOGY FOR NEURAL COMPUTATIONS ON INTEL PROCESSORS

Skribtsov P., skribtsov@physicon.ru,

Science Center of Neurocomputers,
Novaya Basmannaya-20, Moscow, 107066, Russia, scn@mail.cnt.ru

1. Code optimization using SIMD

Special principle called SIMD (Single Instruction Multiple Data) implemented in modern processors' commands extensions under names SSE (Streaming SIMD Extension), 3DNow!, SSE2 is very useful for optimization of neural network applications code. The idea is that one command performs operations with multiple data pieces by means of less number of processor cycles. Here we will talk mostly about SSE standard, which is available on

all Pentium-III compatible machines, including AMD Athlon XP processor.

For example, operation, such as

ADDPS XMM0, XMM1

performs the following actions on the special 128-bit XMM registers that can store 4 single precision floating point values (see Fig. 1)

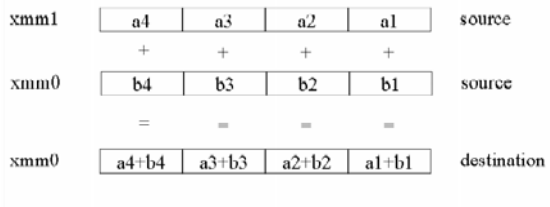

Figure 1.

And it takes only 2 clock cycles!!

This simple picture immediately results in several conclusions:

c) Adjust your network architecture to fit SIMD parameters better. For example, since SSE instructions operate with 4 operands it follows that neurons number in layer should be a multiple of 4. For big enough networks this limitation is not dangerous, because neural networks behave (learn) quite similar if neurons amount vary a little.

d) Adjust your code & data to fit the network architecture better. Use your neural network architecture specifics to optimize the code using SIMD. If network has three inputs (for example if network is processing 3D data), then the first layer has three input-weights and one "free" weight, and it worth to write special code for the first layer that will be working in a different way than for the next layers. This will require to store weights in a different order too (by columns, not rows).

Instead of writing SIMD code by means of assembler language, one can use publicly available libraries, such as Intel® Signal Processing Library 4.5. However, experiments proved that custom code obviously performs better.

The technique described above was used to speed-up Java based neural network framework.

It was tested on AMD Athlon 1.3GHz processor with 133MHz bus. The object oriented framework was written on Java language, with calls to native methods. DLLs for native methods where written in C++ with assembly inlines. To compile SSE assembler instructions Microsoft Visual Studio .NET proved to be convenient. The average performance achieved on this machine after preliminary and deep optimizations happened to be over 2Giga flops/sec.

References

[1] Neural Networks Theory, Galushkin A. I. Radiotechnika,. Moscow 2000

[2] Assembler Special Reference, Victor Yurov, Sankt-Petersburg, 2001, #ISBN 5-272-00119-2

[3] Processors Pentium-4, Athlon and Duron, Michael Guk, Victor Yurov, Sankt-Petersburg, 2001, #ISBN 5-318-00559-4

[4] Intel Performance Library http://developer.intel.com/software/products/perflib/

[5] Example of real time ray-tracing implementation using SSE optimization, Lev Dymchenko, http://www.ixbt.com/video/rt-raytracing.shtml
http://www.virtualray.ru/eng/engine.html