Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Реферат

За первый семестр

По дисциплине: «Теоретические и интеллектуальные информационные технологии»

Тема: «Программирование с CUDA API на C/C++»

Выполнил:

Студент 1 курса

Кирилович А.А.

Проверил:

Анфилец С.В.

Брест 2021

Содержание

Введение

В ноябре 2006 года NVIDIA представила CUDA, программно-аппаратную архитектуру параллельных вычислений, которая использует параллельный вычислитель в графических процессорах NVIDIA для решения множества сложных вычислительных задач более эффективно, чем на CPU.

CUDA поставляется с программной средой, которая позволяет разработчикам использовать C++ как язык программирования высокого уровня. Также CUDA доступна и для других языков программирования таких, как Python и Java.

Преимущества использования графических процессоров и CUDA

- Графический процессор (GPU) обеспечивает гораздо более высокую пропускную способность инструкций и пропускную способность памяти, чем CPU, при аналогичной цене и мощности. Многие приложения используют эти расширенные возможности, чтобы работать на графическом процессоре быстрее, чем на центральном процессоре. Другие вычислительные устройства, такие как FPGA, также очень энергоэффективны, но предлагают гораздо меньшую гибкость программирования, чем графические процессоры.

Эта разница в возможностях GPU и CPU существует потому, что они разработаны с разными целями. В то время как CPU предназначен для максимально быстрого выполнения последовательности операций, называемых *потоками* (англ. *thread*), и может выполнять несколько десятков этих потоков параллельно, графический процессор предназначен для превосходного выполнения тысяч из них параллельно (выполнение одного потока на GPU требует больше времени, чем на CPU, за счет увеличения пропускной способбности).

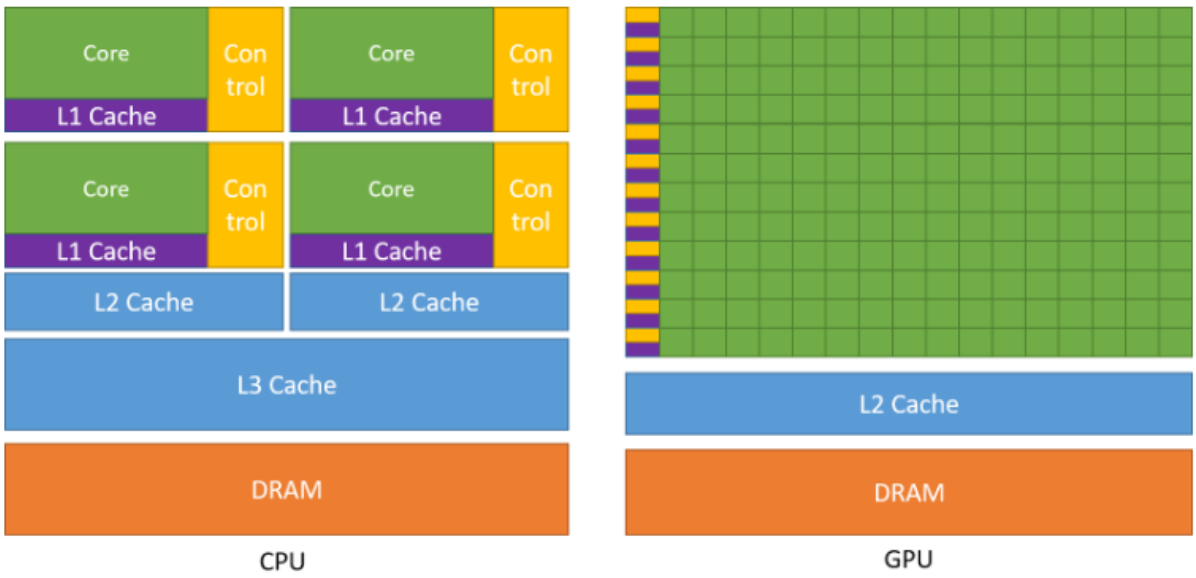


Рисунок 1. Графический процессор выделяет больше транзисторов для обработки данных.

- Графический процессор специализируется на высокопараллельных вычислениях и поэтому спроектирован таким образом, что большинство транзисторов используется для обработки данных, а не для кэширования данных и управления потоком. На схематическом *рисунке 1* показан пример распределения ресурсов микросхемы для центрального процессора в сравнении с графическим процессором.

Выделение большего количества транзисторов для обработки данных, например вычислений с плавающей запятой, полезно для высокопараллельных вычислений; GPU может скрывать задержки доступа к памяти с помощью вычислений, вместо того, чтобы полагаться на большие кеши данных и сложное управление потоком, чтобы избежать длительных задержек доступа к памяти, оба из которых являются дорогостоящими с точки зрения транзисторов.

Как правило, приложение состоит из параллельных и последовательных частей, поэтому системы разрабатываются с сочетанием графических и центральных процессоров, чтобы максимизировать общую производительность. Приложения с высокой степенью параллелизма могут использовать эту массово-параллельную природу графического процессора для достижения более высокой производительности, чем на центральном процессоре.

- Масштабируемая модель программирования

Появление многоядерных CPU и многоядерных GPU означает, что чипы массовых процессоров теперь являются параллельными системами. Задача состоит в том, чтобы разработать прикладное программное обеспечение, которое прозрачно масштабирует свой параллелизм для увеличения числа ядер процессора, подобно тому, как приложения трехмерной графики прозрачно масштабируют свой параллелизм на многоядерные графические процессоры с сильно различающимся числом ядер.

- Модель параллельного программирования CUDA предназначена для решения этой проблемы при сохранении низкой кривой обучения для программистов, знакомых со стандартными языками программирования, такими как C.

В его основе лежат три ключевые абстракции - иерархия групп потоков, разделяемая память и барьерная синхронизация - которые просто открываются программисту как минимальный набор языковых расширений.

Эти абстракции обеспечивают детальный параллелизм данных и параллелизм потоков, вложенный в крупномасштабный параллелизм данных и параллелизм задач. Они помогают программисту разделить проблему на грубые подзадачи, которые могут быть решены независимо параллельно с помощью блоков потоков, и каждую подзадачу на более мелкие части, которые могут быть решены совместно, параллельно всеми потоками в блоке.

Эта декомпозиция сохраняет выразительность языка, позволяя потокам взаимодействовать при решении каждой подзадачи, и в то же время обеспечивает автоматическую масштабируемость. Действительно, каждый блок потоков может быть запланирован на любом из доступных мультипроцессоров в GPU в любом порядке, одновременно или последовательно, так что скомпилированная программа CUDA может выполняться на любом количестве мультипроцессоров, как показано на *рисунке 2*.

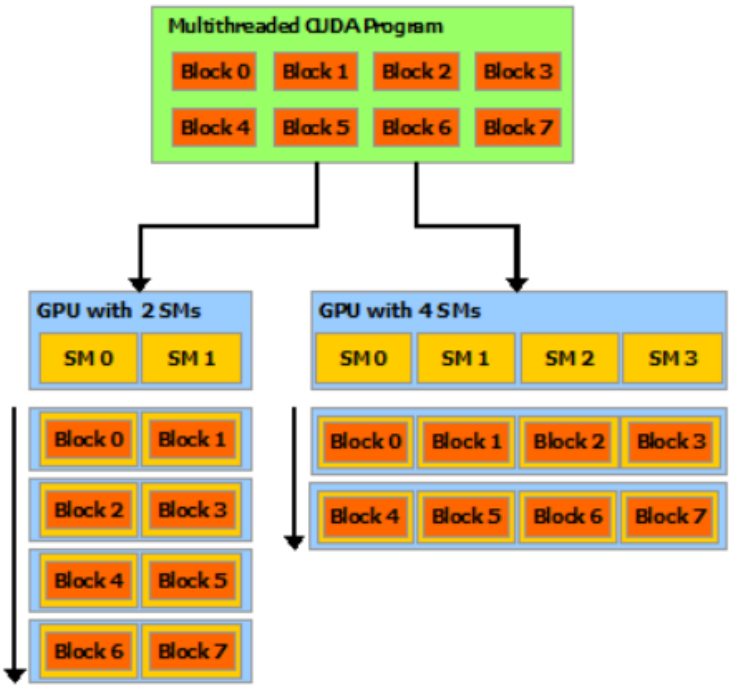


Рисунок 2. Автоматическая масштабируемость

Примечание. Графический процессор построен на основе массива потоковых мультипроцессоров (англ. Streaming Multiprocessors). Многопоточная программа разбивается на блоки потоков, которые выполняются независимо друг от друга, поэтому графический процессор с большим количеством мультипроцессоров автоматически выполняет программу за меньшее время, чем графический процессор с меньшим количеством мультипроцессоров.

Эта масштабируемая модель программирования позволяет архитектуре графического процессора охватить широкий диапазон рынка путем простого масштабирования количества мультипроцессоров и разделов памяти: от высокопроизводительных графических процессоров GeForce для энтузиастов и профессиональных вычислительных продуктов Quadro и Tesla до различных недорогих массовых графических процессоров GeForce.

Модель программирования

В этой главе представлены основные концепции, лежащие в основе модели программирования CUDA, и показано, как они представлены в C++.

- **Ядра**

CUDA C ++ расширяет C ++, позволяя программисту определять функции C++, называемые *ядрами* (англ. *kernels*), которые при вызове выполняются N раз параллельно N различными *потоками CUDA*, а не только один раз, как обычные функции C++.

Ядро определяется с помощью `__global__` спецификатор объявления и количество потоков CUDA, которые выполняют это ядро для данного вызова ядра, указываются с использованием нового `<<< ... >>>` синтаксис *конфигурации выполнения*. Каждому потоку, выполняющему ядро, дается уникальный *идентификатор потока,* который доступен в ядре через встроенные переменные.

В качестве иллюстрации следующий пример кода с использованием встроенной переменной `threadIdx` складывает два вектора *A* и *B* размера *N* и сохраняет результат в вектор *C*:

```
// Определение ядра

__global__ void VecAdd (float * A, float * B, float * C) {

int i = threadIdx .x;

    C[я] = A[я] + B[я];

}

int main () {

    ... // Вызов ядра с N потоками

    VecAdd <<< 1, N >>> (A, B, C);

    ...

}
```

Здесь каждый из *N* потоков, выполняющих `VecAdd ()` выполняет одно попарное сложение.

- **Иерархия потоков**

Для удобства, `threadIdx` представляет собой трехкомпонентный вектор, так что потоки (в некоторых книгах их называют нитями, но здесь и далее будем употреблять термин потоки) могут быть идентифицированы с использованием одномерного, двухмерного или трехмерного *индекса потока*, образуя одномерный, двухмерный или трехмерный блок потоков, называемый *блок потоков*. Это

обеспечивает естественный способ вызова вычислений для элементов в домене, таком как вектор, матрица или объем.

Индекс потока и его идентификатор связаны друг с другом напрямую: для одномерного блока они одинаковы; для двумерного блока размера (D_x, D_y) идентификатор потока для потока с индексом (x, y) равен $(x + y D_x)$; для трехмерного блока размером (D_x, D_y, D_z) идентификатор потока для потока с индексом (x, y, z) равен $(x + y D_x + z D_x D_y)$.

В качестве примера следующий код добавляет две матрицы *A* и *B* размером *NxN* и сохраняет результат в матрицу *C*:

```
// Определение ядра

__global__ void MatAdd (float A[N][N], float B[N][N],

                        float C[N][N]) {

    int i = threadIdx.x;

    int j = threadIdx.y;

    C[i][j] = A[i][j] + B[i][j];

}

int main () {

    ... // Вызов ядра с одним блоком из N * N * 1 потоков

    int numBlocks = 1;

    dim3 ThreadsPerBlock (N, N);

    MatAdd <<< numBlocks, threadPerBlock >>> (A, B, C);

    ...

}
```

Существует ограничение на количество потоков на блок, поскольку ожидается, что все потоки блока будут находиться в одном ядре процессора и должны совместно использовать ограниченные ресурсы памяти этого ядра. На текущих графических процессорах блок потока может содержать до 1024 потоков.

Однако ядро может выполняться несколькими блоками потоков одинаковой формы, так что общее количество потоков равно количеству потоков в блоке, умноженному на количество блоков.

Блоки организованы в одномерную, двухмерную или трехмерную *сетку* блоков резьбы, как показано на *рисунке 3*. Количество блоков потоков в сетке обычно

определяется размером обрабатываемых данных, который обычно превышает количество процессоров в системе.

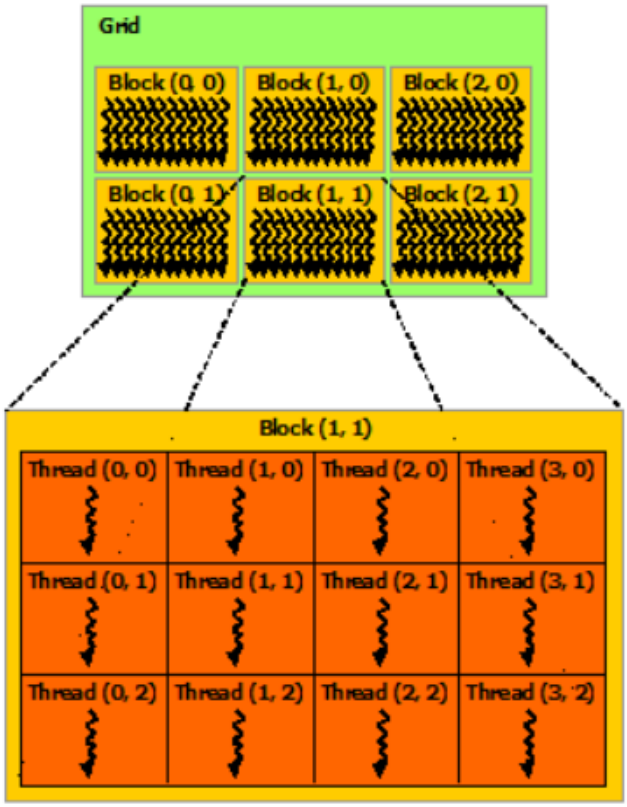


Рисунок 3. Сетка блоков потоков

Количество потоков на блок и количество блоков на сетку, указанные в `<<<...>>>` синтаксис может иметь тип `int` или `dim3`. Двумерные блоки или сетки могут быть указаны, как в примере выше.

Каждый блок в сетке можно идентифицировать по одномерному, двумерному или трехмерному уникальному индексу, доступному в ядре через встроенную переменную `blockIdx`. Размер блока потока доступен в ядре через встроенную переменную `blockDim`.

Несколько изменяем предыдущий пример с `MatAdd ()` для обработки нескольких блоков - код становится следующим:

```
// Определение ядра

__global__ void MatAdd(float A[N][N], float B[N][N],

float C[N][N]){

    int i = blockIdx.x * blockDim.x + threadIdx.x;

    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < N && j < N)
```

```
        C[i][j] = A[i][j] + B[i][j];
}
int main(){
    ...
    // Вызов ядра
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Размер блока потоков - 16x16 (256 потоков), хотя и произвольный в данном случае, является обычным выбором. Сетка создается с таким количеством блоков, чтобы, как и раньше, иметь один поток на элемент матрицы. Для простоты в этом примере предполагается, что количество потоков на сетку в каждом измерении равномерно делится на количество потоков на блок в этом измерении, хотя это не обязательно.

Блоки потоков должны выполняться независимо: они должны выполняться в любом порядке, параллельно или последовательно. Это требование независимости позволяет планировать блоки потоков в любом порядке для любого количества ядер, что позволяет программистам писать код, масштабируемый с увеличением количества ядер.

Потоки внутри блока могут взаимодействовать, обмениваясь данными через некоторую *общую память* и синхронизируя их выполнение для координации доступа к памяти. Точнее, можно указать точки синхронизации в ядре, вызвав внутреннюю функцию `__syncthreads();`

`__syncthreads()` действует как барьер, на котором все потоки в блоке должны ждать, прежде чем любой из них будет разрешен для продолжения. Общая память дает пример использования общей памяти.

- **Иерархия памяти**

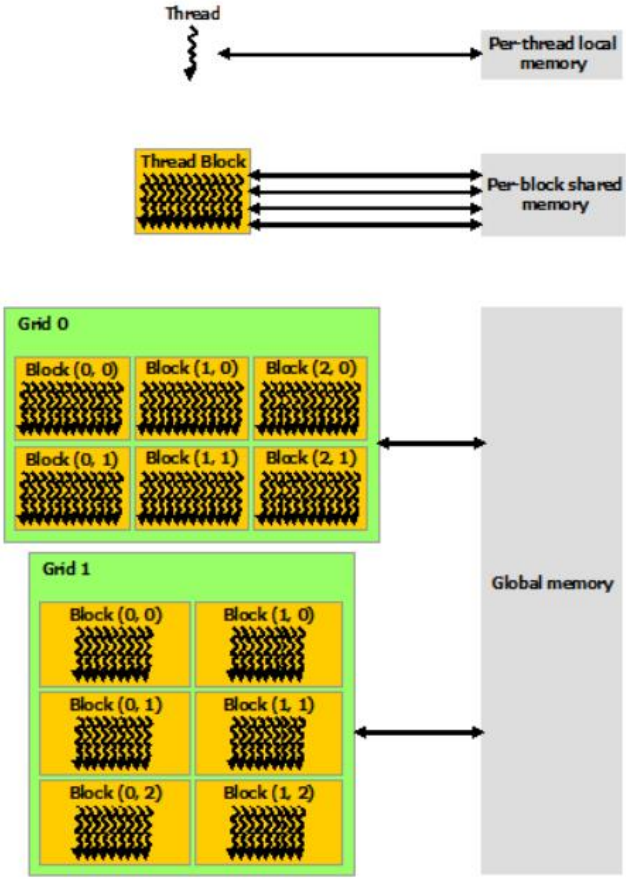
Потоки CUDA могут получать доступ к данным из нескольких пространств памяти во время своего выполнения, как показано на *рисунке 4*. У каждого потока есть собственная локальная память. Каждый блок потока имеет общую память, видимую для всех потоков блока, и имеет то же время жизни, что и блок. Все потоки имеют доступ к одной и той же глобальной памяти.

Также есть два дополнительных пространства памяти только для чтения, доступных для всех потоков: пространство констант и памяти текстуры. Пространства глобальной, константной и текстурной памяти

оптимизированы для различного использования памяти. Память текстур также предлагает различные режимы адресации, а также фильтрацию данных для некоторых конкретных форматов данных.

Пространства глобальной, константной и текстурной памяти сохраняются при запуске ядра одним и тем же приложением.

Рисунок 4. Иерархия памяти



- **Гетерогенное программирование**

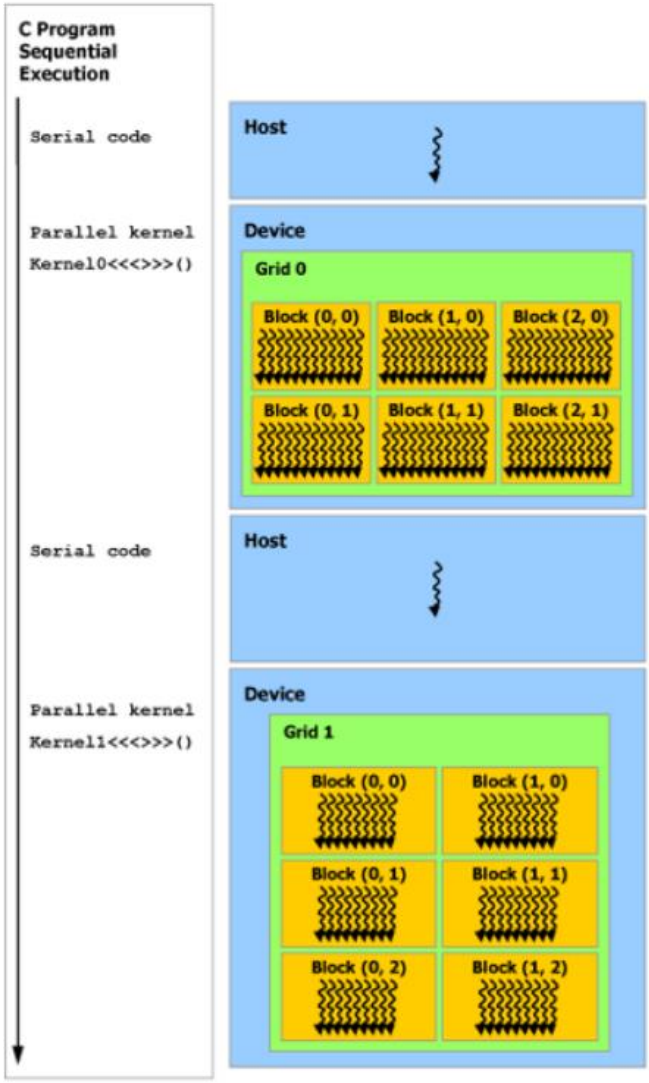
Как показано на *рисунке 5*, модель программирования CUDA предполагает, что потоки CUDA выполняются на физически отдельном *устройстве,* которое работает как сопроцессор по отношению к *хосту,* на *котором* выполняется программа C++. Это имеет место, например, когда ядра выполняются на графическом процессоре, а остальная часть программы C++ выполняется на процессоре.

Модель программирования CUDA также предполагает, что оба хоста и устройство поддерживают свои собственные отдельные пространства памяти в DRAM, называемых *памятью хоста* и *памятью устройства*, соответственно. Следовательно, программа управляет пространством глобальной, константной и текстурной памяти, видимым ядрам посредством вызовов среды

выполнения CUDA. Это включает в себя выделение и освобождение памяти устройства, а также передачу данных между хостом и памятью устройства.

Unified Memory предоставляет *управляемую память* для объединения пространств памяти хоста и устройства. Управляемая память доступна для всех процессоров и графических процессоров в системе как единый согласованный образ памяти с общим адресным пространством. Эта возможность допускает превышение лимита памяти устройства и может значительно упростить задачу переноса приложений, устраняя необходимость в явном зеркальном отображении данных на хосте и устройстве.

Рисунок 5. Гетерогенное программирование



Примечание. Последовательный код выполняется на хосте, а параллельный код выполняется на устройстве.

- **Модель асинхронного программирования**

В модели программирования CUDA поток - это самый низкий уровень абстракции для выполнения вычислений или операций с памятью. Начиная с устройств на базе архитектуры NVIDIA Ampere GPU, модель программирования CUDA обеспечивает

ускорение операций с памятью с помощью модели асинхронного программирования. Модель асинхронного программирования определяет поведение асинхронных операций по отношению к потокам CUDA.

Асинхронная операция определяется как операция, которая инициируется потоком CUDA и выполняется асинхронно как если бы другим потоком. В правильно сформированной программе один или несколько потоков CUDA синхронизируются с асинхронной операцией. Поток CUDA, инициировавший асинхронную операцию, не обязательно должен быть среди синхронизирующих потоков.

Такой асинхронный поток («as-if thread») всегда связан с потоком CUDA, который инициировал асинхронную операцию. Асинхронная операция использует объект синхронизации для синхронизации завершения операции. Таким объектом синхронизации может явно управлять пользователь (например, cuda::memcpy_async) или неявно управляемый в библиотеке (например, cooperative_groups::memcpy_async).

Применение

Так как многопоточное программирование полезно для решение большого числа монотонных задач, то оно находит применение во многих сферах. Например, рендеринг изображений, обучение нейронных сетей, решение сложных математических и физических задач. Если рассматривать каждую область, то это займет очень много страниц текста, так что я рассмотрю лишь одну – обучение нейронных сетей:

- Механизм обучения искусственных нейронных сетей (ИНС) с помощью CUDA

Механизм распараллеливания обучения ИНС на основе алгоритма обратного распространения ошибки основан на пошаговом расчете параметров всех слоев сети, так как, не зная выходных параметров предыдущего (в режиме прогнозирования) или последующего (при обучении) слоя, невозможно вычислить параметры текущего слоя . Существует два варианта распараллеливания данного алгоритма:

1) одновременное обучение нескольких ИНС с последующим выбором наиболее оптимальной из них по критерию минимальной ошибки;
2) одновременный расчет параметров нескольких нейронов одного слоя.

Для обучения ИНС был выбран второй способ. Он требует выделения на устройстве меньшего количества памяти, чем первый, что позволяет применять его для ИНС с большим количеством нейронов в слоях.

Обучение ИНС может быть представлено в виде трех этапов:

1) прямой проход (вычисление выходов каждого слоя, начиная с первого);
2) вычисление ошибки слоев, начиная с последнего;
3) вычисление корректирующих значений для матриц коэффициентов, векторов порогов и изменение текущих матриц весов и векторов порогов с учетом рассчитанных значений.

Из-за аппаратных особенностей видеокарт количество нейронов в слое должно быть кратно 2^n, где $n = \{1, \ldots, 5\}$. Соответственно для сетей с количеством нейронов по слоям больше 8 число нейронов должно быть кратно 16, для чего матрицы весов (W_i), вектора входов (Y_{i-1}), выходов (Y_i) и порогов (T_i) выравниваются (заполняются нулями) до выбранных размерностей.

Схема взаимодействия внешнего и разработанного (в виде библиотеки) программных модулей (далее – модулей) представлена на *рисунке 6*. Исходные данные по состоянию сети (W_i , ΔW_i , T_i , ΔT_i, E_i , Y_i для каждого слоя) формируются внешним модулем. Если сеть обучается впервые, то W_i и T_i заполняются случайными числами в диапазоне (0; 1) или используются иные

алгоритмы (например, генетические). Корректирующие значения весов и порогов (ΔW_i и ΔT_i), текущие значения вектора ошибки (E_i) и вектора выходов слоя (Y_i) заполняются нулями. Если сеть дообучается, то используются ранее сохраненные значения W_i , ΔW_i , T_i , ΔT_i , E_i , Y_i.

Указанные параметры передаются по ссылке на управляющий модуль (host). Под них выделяется память на устройстве (device). Также выделяется память для массивов, необходимых при промежуточных расчетах. Наборы обучающих данных (Inputs, Outputs), входящих в одну эпоху, передаются в начале обучения с внешнего модуля на управляющий, а затем на устройство.

Расчет включает в себя последовательное выполнение прямого и обратного проходов для каждого набора входных и выходных данных (Inputs, Outputs), составляющих одну эпоху. При этом на управляющий модуль с устройства копируется значение среднеквадратического отклонения (СКО) выходного слоя, используемое при вычислении суммарного СКО каждой эпохи, записываемого в файл. По СКО эпохи можно судить о том, насколько обучена сеть. По окончании обучения (достижения определенного уровня СКО или выполнения заданного количества итераций) устройство возвращает данные по состоянию сети W_i , ΔW_i , T_i , ΔT_i , E_i , Y_i для каждого слоя обратно управляющему модулю, а тот в свою очередь – внешнему модулю.

Таким образом, исключается необходимость постоянного перемещения требуемых для обучения параметров между устройством и управляющем модулем. Однажды получив их, устройство само обучает сеть, периодически возвращая только значение СКО для обеспечения возможности отслеживания текущего состояния ИНС. При необходимости W_i и T_i для каждого слоя могут

быть переданы для последующего использования во внешних графических модулях, визуализирующих текущее состояние ИНС (см. рис. 2). Однако это увеличивает время обучения. Графическое отображение состояния сети целесообразно использовать на этапе определения ее топологии, когда визуализация параметров позволяет изменять структуру для минимизации СКО.

- Тестирование программного модуля

Тестирование скорости работы разработанного модуля проводилось на ЭВМ, оснащенной 4-ядерным процессором Intel Core 2 Quard (по 2,86 ГГц) и видеокартой GeForce 9500 GT, а также на видеокартах GeForce 9800 GT и GeForce 220 M GT, параметры которых приведены в *табл. 1*.

Важную роль в быстродействии работы CUDA-совместимого устройства играет версия драйверов, обеспечивающих программный доступ к нему (см. *табл. 2*). В *табл. 2* приведены средние замеры времени в миллисекундах, затраченного на обучение одного скрытого слоя за одну эпоху на одном и том же устройстве, но с разной версией драйверов (2.3 и 3.0). Количество входных сигналов для каждого нейрона равно количеству нейронов в рассматриваемом слое (…–N_{i-1}–N_i–…), обучающая выборка включала 1000 наборов входных данных. Прирост производительности версии 3.0 по сравнению с версией 2.2 составляет около 9%. На рис. 4 и рис. 5 показаны графики затраченного времени в секундах на обучение выровненной ИНС с двумя скрытыми слоями (N – 2N – N) на 10 и 1000 наборах данных соответственно за 10 итераций.

Таблица 1. Параметры тестируемых устройств

$N_i=N_{i-1}$	Версия драйверов		Выигрыш во времени, %	$N_i=N_{i-1}$	Версия драйверов		Выигрыш во времени, %
	2.3, мс	3.0, мс			2.3, мс	3.0, мс	
32	140	110	21,4	288	2859	2656	7,1
64	172	157	8,7	320	3813	3531	7,4
96	266	250	6,0	352	4954	4563	7,9
128	437	390	10,7	384	6281	5781	8,0
160	688	625	9,1	416	7875	7250	7,9
192	1000	937	6,3	448	9688	8922	7,9
224	1515	1391	8,2	480	11844	10859	8,3
256	2094	1938	7,4	512	14094	12969	8,0

Таблица 2. Время обучения одного слоя ИНС на видеокартах с разными версиями драйверов

Модель GeForce	Дата выхода на рынок	Объем видеопамяти (Мб)	Конфигурация ядра *	Частоты			Пиковая скорость заполнения		Память			Теоретическая производительность (Гигафлопс)
				Ядро,(MHz)	Шейдерный блок, (MHz)	Память (Mhz)	Гигатекселей/с	Гигатекселей/с	Пропускная способность (Гб/с)	Тип	Шина (бит)	
220M GT	16.08 2009	1024	32: 16:08	500	1250	1000 1600	4	8	16	DDR2	128	120
9500 GT	29.07 2008	256	32: 16:08	550	1400	1000 1600	4,4	8,8	16 25,6	DDR2 GDDR3	128	134
9800 GT	август 2008	512	112: 56:16	600	1500	1800	9,6	33,6	57,6	GDDR3	256	504

Рис. 7. Затраченное время на обучение ИНС с двумя скрытыми слоями (эпоха из 10 наборов)

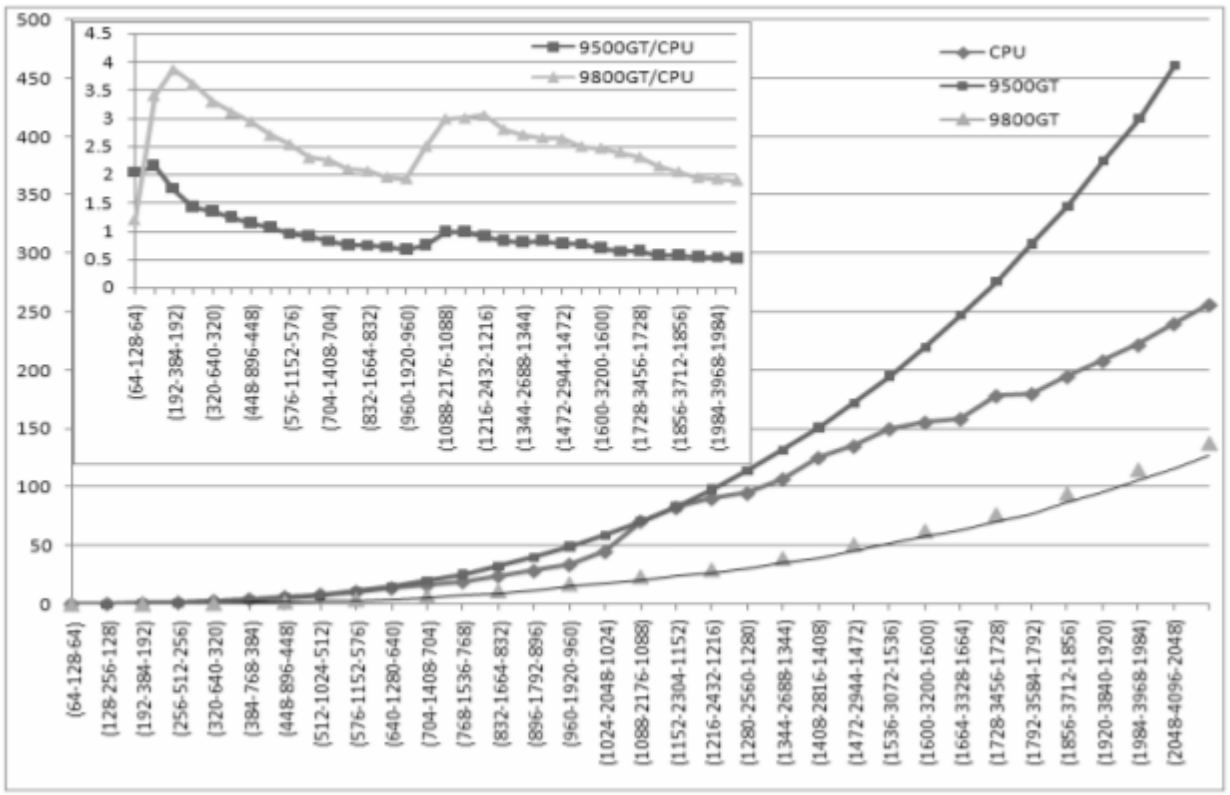
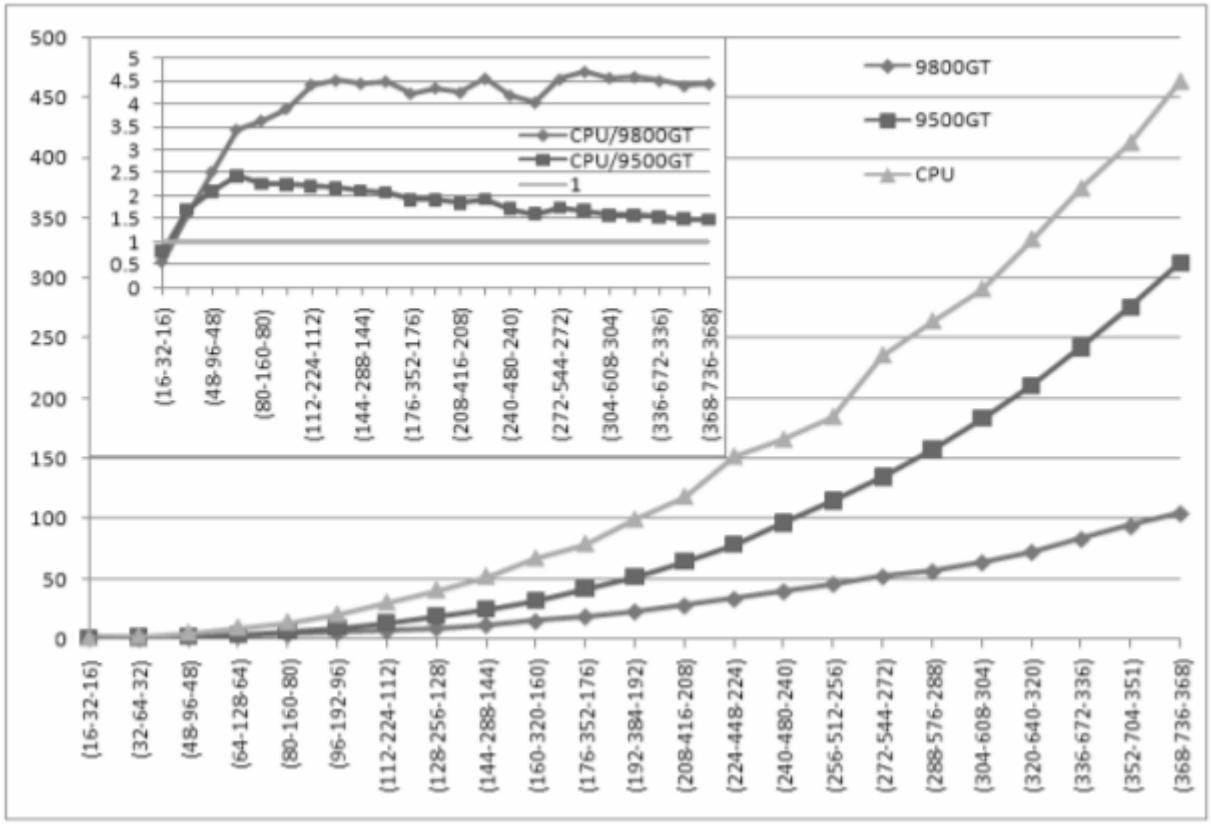


Рис. 5. Затраченное время на обучение ИНС с двумя скрытыми слоями (эпоха из 1000 наборов)



На малых графиках (*рис. 7* и *рис. 8*) показано, во сколько раз быстрее по сравнению с центральным процессором (CPU) обучается ИНС указанной на оси абсцисс топологии на видеокартах GeForce 9800 GT и 9500 GT. Используя CUDA-технологию, удалось ускорить обучение ИНС по сравнению с CPU до 5 раз на 9800 GT и до 2,5 раз на 9500 GT. Время обучения включает в себя время работы с файлами и минимальное перемещение данных между host и device. Выигрыш наблюдается уже в сетях (32 – 64 – 32). При обучении на 1000 наборах данных наибольший выигрыш на 9500 GT достигается при обучении ИНС (64 – 128 – 64), на 9800 GT – при обучении ИНС (288 – 576 – 288).

Установка средств разработки CUDA

Чтобы использовать CUDA в вашей системе, вам необходимо установить следующее:

- Графический процессор с поддержкой CUDA
- Поддерживаемая версия Microsoft Windows
- Поддерживаемая версия Microsoft Visual Studio
- Набор инструментов NVIDIA CUDA (доступен по адресу http://developer.nvidia.com/cuda-downloads

Основные инструкции можно найти на сайте https://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html#windows.

Настройка средств разработки CUDA в системе с соответствующей версией Windows состоит из нескольких простых шагов:

1)Убедитесь, что в системе есть графический процессор с поддержкой CUDA.

2)Загрузите NVIDIA CUDA Toolkit.

3)Установите NVIDIA CUDA Toolkit.

4)Убедитесь, что установленное программное обеспечение работает правильно и взаимодействует с оборудованием.

1. Убедитесь, что у вас есть графический процессор с поддержкой CUDA

Если у вас есть процессор NVIDIA, этот графический процессор поддерживает CUDA.

2. Загрузите NVIDIA CUDA Toolkit

Набор инструментов NVIDIA CUDA доступен по адресу http://developer.nvidia.com/cuda-downloads . Выберите платформу, которую вы используете, и один из следующих форматов установщика:

1. Сетевой установщик: минимальный установщик, который позже загружает пакеты, необходимые для установки. Загружаются только пакеты, выбранные на этапе выбора установщика. Этот установщик полезен для пользователей, которые хотят минимизировать время загрузки.
2. Полный установщик: установщик, который содержит все компоненты CUDA Toolkit и не требует дополнительной загрузки. Этот установщик полезен для систем, в которых отсутствует доступ к сети, и для развертывания на предприятии.

CUDA Toolkit устанавливает драйвер CUDA и инструменты, необходимые для создания, сборки и запуска приложения CUDA, а также библиотеки, файлы заголовков, образцы исходного кода CUDA и другие ресурсы.

3. Установите программное обеспечение CUDA

Перед установкой инструментария вам следует
прочитать *примечания* к *выпуску* CUDA Toolkit, поскольку они содержат подробную информацию об установке и функциональных возможностях программного обеспечения.

Есть два варианта:

- Установка графически

  Установите программное обеспечение CUDA, запустив установщик CUDA и следуя подсказкам на экране.

- Установка с использованием Conda для установки программного обеспечения CUDA

  Пакеты Conda доступны по адресу https://anaconda.org/nvidia .

  Чтобы выполнить базовую установку всех компонентов CUDA Toolkit с помощью Conda, нужно выполнить следующую команду:

  conda install cuda -c nvidia

Более подробно об установке нужно смотреть в инструкциях от NVIDIA.

Заключение

При исследовании такой технологии, как технология CUDA, можно понять, что она действительно может ускорить некоторые вычислительные процессы в несколько раз. Однако не стоит ее использовать там, где это не необходимо, так как эти вычисления вообще могут замедлиться, а не ускориться. Благодаря технологии CUDA можно использовать в решении задач не только CPU, но и GPU, причем очень продуктивно. Единственным, наверно, ограничением для использования этой технологии является само отсутствие специализированных графический процессоров, то есть процессоров NVIDIA, в компьютере. Однако и эту проблему можно решить, если использовать какой-нибудь облачный сервис с поддержкой вышеупомянутого процессора. Подводя итог, можно сказать, что в целом технология CUDA хороша (в умелых руках конечно же) во многих отношениях, и, если и имеет недостатки, то незначительные.

Литература

1. https://habr.com/ru/company/epam_systems/blog/245503/
2. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model
3. https://habr.com/ru/post/424845/
4. https://habr.com/ru/post/393071/
5. https://habr.com/ru/news/t/371917/
6. https://habr.com/ru/post/470742/
7. https://russianblogs.com/article/6938518331/
8. https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch01.html
9. http://www.mathnet.ru/links/ba81ceb946723000b547f7e0d2aa0d0a/vsgtu990.pdf
10. https://cyberleninka.ru/article/n/primenenie-cuda-tehnologii-dlya-obucheniya-iskusstvennyh-neyronnyh-setey/viewer
11. https://cyberleninka.ru/article/n/primenenie-tehnologii-cuda-dlya-uskoreniya-vychisleniy-v-neyronnyh-setyah/viewer

Дата посещения всех сайтов использованных для составления реферата 30.12.2021