УДК 004.052.32 DOI: 10.31114/2078-7707-2022-4-94-100

Аппаратная реализация ускоренного приближённого матричного умножителя на основе алгоритма MADDNESS

А.С. Жигулин, Р.А. Соловьев

Институт проблем проектирования в микроэлектронике РАН, г. Москва, lenshtein-z@yandex.ru

***Аннотация* — Предложена методика по созданию аппаратной реализации ускоренного приближённого матричного умножителя MADDNESS. Данный умножитель имеет хорошие показатели по точности и скорости работы и одновременно отличается простотой декодера, что позволяет его широко применять в аппаратной реализации нейронных сетей. В результате исследований удалось достичь очень высокой скорости работы умножителя на аппаратном уровне за счёт полного отказа от операции умножения как таковой. При этом качество полученных предсказаний остаётся высоким.**

***Ключевые слова* — аппаратная реализация матричного умножителя, Программируемые Логические Интегральные Схемы, целочисленная арифметика.**

I. Введение

Операция матричного умножения лежит в основе работы большинства нейронных сетей. Сложность операции, применяемой большинством математических пакетов, $O(n^3)$, что ограничивает размеры сетей и заставляет использовать высокопроизводительное оборудование для их работы. Однако для нейронных сетей не требуется производить точное умножение, поэтому было разработано немало методов приближённого матричного умножения, которое значительно ускоряет работу.

Задача ускоренного матричного умножения формулируется следующим образом: мы имеем матрицы $\boldsymbol{A} \in \mathbb{R}^{N\times D}, \boldsymbol{B} \in \mathbb{R}^{D\times M}$, произведение которых $\boldsymbol{AB} \in \mathbb{R}^{N\times M}$ мы ищем (рис. 1), причём $N \gg D$. Имея время вычисления τ, требуется построить функции $g(\cdot),\ h(\cdot)$ и $f(\cdot,\cdot)$ и найти константы $\alpha, \boldsymbol{\beta}$ такие, что

$$\left\|\alpha f\big(g(\boldsymbol{A}), h(\boldsymbol{B})\big) + \boldsymbol{\beta} - \boldsymbol{AB}\right\|_F < \varepsilon(\tau)\|\boldsymbol{AB}\|_F$$

с как можно меньшей относительной погрешностью ε(τ).

Есть два класса ускоренных матричных умножителей. Первый основан на снижении размерности требуемого умножения с помощью промежуточных умножений на специальные матрицы:

$$\boldsymbol{AB} \approx (\boldsymbol{AV_A})(\boldsymbol{V_B^T B}), \boldsymbol{V_A}, \boldsymbol{V_B} \in \mathbb{R}^{D\times d}, d \ll D.$$

На линейных преобразованиях подобного вида построены такие методы, как PCA, SparcePCA [1], FastJL [2] и HashJL [3].

Второй основан на нелинейном классификаторе $g(\cdot)$ и таблице поиска $f\big(\cdot, h(\cdot)\big)$. К ним относятся ScalarQuantize, Product Quantization [4] и основанные на последнем Bolt [5] и MADDNESS [6]. Скорость и точность работы таких умножителей в значительной степени зависит как от типа применённого классификатора, так и от настраиваемых параметров, и может быть как медленнее линейного умножителя, так и быстрее при сравнимой точности. При этом оба класса умножителей требуют предварительного обучения, так как они подстраиваются под определённый набор входных данных.

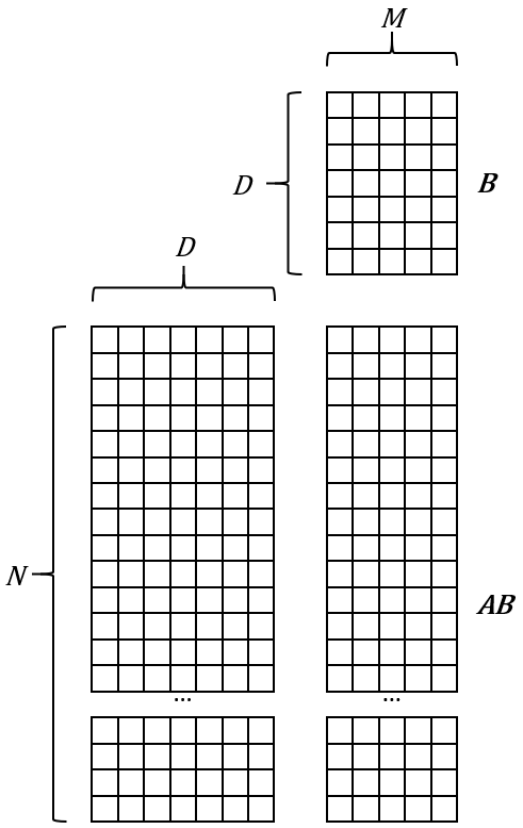

Рис. 1. Входные (A, B) и выходная (AB) матрицы

II. Описание умножителя MADDNESS

Умножитель MADDNESS был предложен в 2021 году Дэвисом Блалоком и Джоном Гуттагом [6]. Он базируется на предположении $\boldsymbol{a}^T\boldsymbol{b} \approx \hat{\boldsymbol{a}}^T\boldsymbol{b}$ ($\boldsymbol{a}^T$ – строка

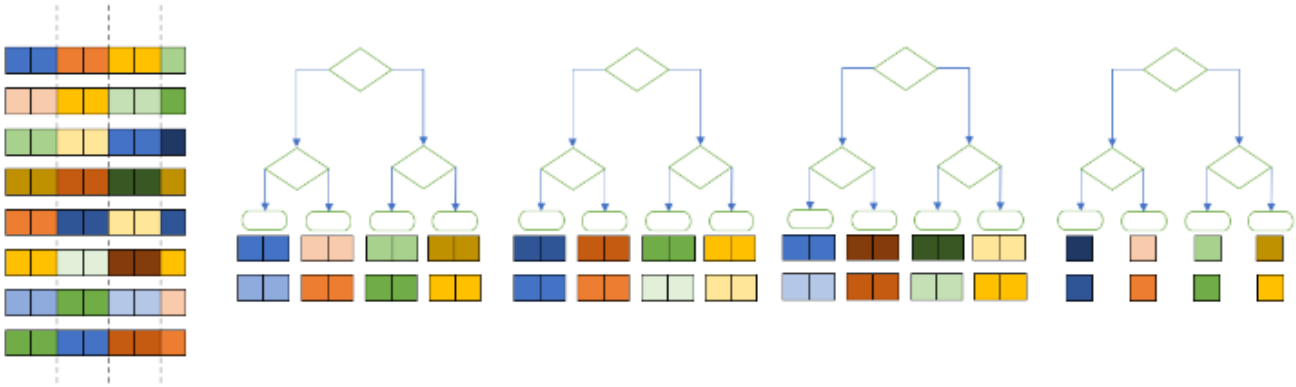

Рис. 2. Разбиение строк тренировочной матрицы на подстроки и распределение внутри деревьев. Элементы, относящиеся к разным деревьям-классификаторам, на левом рисунке отделены друг от друга пунктиром, каждый набор подстрок слева относится к своему дереву справа. Разными оттенками одного цвета обозначены похожие подстроки

матрицы $\boldsymbol{A}$ размера $N \times 1$, $\boldsymbol{b}$ – столбец матрицы $\boldsymbol{B}$ размера $1 \times N$), где $\|\boldsymbol{a} - \hat{\boldsymbol{a}}\|$ мало. При этом $\hat{\boldsymbol{a}}$ является конкатенацией нескольких векторов-прототипов, полученных во время этапа обучения. Основное ускорение происходит за счёт того, что мы заранее вычисляем $\hat{\boldsymbol{a}}^T \boldsymbol{b}$ и в процессе вычислений подставляются необходимые результаты.

Алгоритм MADDNESS разбивается на следующие этапы:

1. Определение подпространств из подстрок – определяется C наборов индексов (номеров элементов строки), внутри которых будут строиться деревья-классификаторы. Обычно разбиение идёт на равные идущие подряд части (рис. 2 слева).
2. Обучение классификатора $g(\cdot)$ – построение сбалансированных бинарных регрессионных деревьев для каждого из C непересекающихся подпространств из подстрок, на которые разбивается строка $\boldsymbol{a}$ (рис. 2 справа). Каждый лист этих деревьев соответствует определённому набору подстрок из тренировочной матрицы, а каждый узел – сравнению элемента строки с некоторым порогом.
3. Построение прототипов подстрок – подбор подстрок, по которым можно восстановить тренировочную выборку с минимальной погрешностью. Эти подстроки ставятся в соответствие с листьями полученных в предыдущем этапе деревьев.
4. Создание таблицы поиска $h(\boldsymbol{B})$ – предвычисление скалярных произведений $\boldsymbol{b}$ и каждого из прототипов, результаты связываются с листьями полученных на первом этапе деревьев.
5. Применение кодирующей функции $g(\boldsymbol{a})$ – разбиение строки на подстроки и определение соответствия прототипам. Каждой из подстрок при этом присваивается соответствующий индекс.
6. Агрегация $f(g(\boldsymbol{a}), h(\boldsymbol{B}))$ – использование полученных индексов и предвычисленных произведений для получения частичных скалярных произведений и их последующее сложение.

В итоге алгоритм разделяется на 2 крупных этапа: этап обучения (ищем, как дробить строки – учим регрессионные деревья на разделённой тренировочной матрице – ищем прототипы подстрок, на которые похожи результаты классификации тренировочной матрицы – ищем произведения прототипов на вторую матрицу), который не требует аппаратной реализации, и рабочий этап (применение регрессионных деревьев – сложение их результатов), аппаратная реализация которого и создаётся. При этом если для этапа обучения нам необходима вся тренировочная матрица сразу, то рабочий этап может обрабатывать первую матрицу построчно, так как на этом этапе строки никак друг на друга не влияют. Подробнее эти этапы будут разобраны далее.

III. Обучение классификатора и кодирующая функция

В качестве кодирующей функции авторами алгоритма было выбрано сбалансированное бинарное регрессионное дерево. Каждый лист этого дерева будет соответствовать своему набору подстрок. Решение на каждом узле дерева принимается путём сравнения одного из значений подстроки с эталоном.

Для обеспечения SIMD-инструкций авторами алгоритма размер дерева был ограничен 16 листьями. Для микросхем, построенных специально для этого алгоритма, такие ограничения неактуальны, поэтому пришлось изменить программный код, реализующий его. Также авторами алгоритма было установлено ограничение «один уровень – один индекс» по тем же причинам.

Таким образом, для записи одного дерева достаточно использовать набор из n индексов

(номеров элементов строки, по которые сравниваются с пороговыми значениями) $j^1,\dots,j^n$ и n наборов пороговых значений $\boldsymbol{v}^1,\dots,\boldsymbol{v}^n$, причём набор $\boldsymbol{v}^t$ имеет размет 2^{t-1}.

Сам по себе алгоритм классификации легко реализуется в ПЛИС и ASIC и не требует каких-либо умножителей:

Вход: вектор $\boldsymbol{x}$, индексы сечения $j^1,\dots,j^n$, пороги сечения $\boldsymbol{v}^1,\dots,\boldsymbol{v}^n$
$i \leftarrow 1$
цикл $t \leftarrow 1$ **до** n:
    $v \leftarrow \boldsymbol{v}_i^t$
    $b \leftarrow x_{j^t} \geq v ? 1 : 0$
    $i \leftarrow 2i - 1 + b$
конец цикла
вернуть i

Индексы и пороги сечения обучены на обучающей матрице $\widehat{\boldsymbol{A}}$ с помощью жадного алгоритма построения дерева. Для описания алгоритма, необходимо ввести определение *бакета* $\mathcal{B}_i^t$ – набора векторов, отнесённых к узлу i на уровне t дерева. Таким образом, корнем дерева будет бакет $\mathcal{B}_1^0$, содержащий все вектора.

Алгоритм по поиску лучшего разбиения на конкретном этапе:

Вход: бакеты $\mathcal{B}_1^{t-1},\dots,\mathcal{B}_{2^{t-1}}^{t-1}$, набор индексов $J^{(c)}$
$l^{min}, j^{min}, \boldsymbol{v}^{min} \leftarrow \infty, NaN, NaN$
цикл $j \in J^{(c)}$:
    $l \leftarrow 0$
    $\boldsymbol{v} \leftarrow [\,]$
    цикл $i \leftarrow 1$ до 2^{t-1}:

        $v_i, l_i \leftarrow$ оптимальное разбиение$(j, \mathcal{B}_i^{t-1})$
            присоединить$(\boldsymbol{v}, v_i)$
            $l \leftarrow l + l_i$
    конец цикла
    если $l < l^{min}$:
        $l^{min} \leftarrow l, j^{min} \leftarrow j, \boldsymbol{v}^{min} \leftarrow \boldsymbol{v}$
    конец если
конец цикла
$\boldsymbol{\mathcal{B}} \leftarrow [\,]$
цикл $i \leftarrow 1$ до 2^{t-1}:
    $\mathcal{B}_{\text{меньше}}, \mathcal{B}_{\text{больше}} \leftarrow$
подтвердить разделение$\left(v_i^{min}, j^{min}, \mathcal{B}_i^{t-1}\right)$
    присоединить$(\boldsymbol{\mathcal{B}}, \mathcal{B}_{\text{меньше}})$
    присоединить$(\boldsymbol{\mathcal{B}}, \mathcal{B}_{\text{больше}})$
конец цикла
вернуть $\boldsymbol{\mathcal{B}}, l^{min}, j^{min}, \boldsymbol{v}^{min}$

Оптимальное разбиение по индексу ищется следующим образом: строки в бакете сортируются по столбцу j. Для каждой строки берётся разбиение бакета на бакет со строками до этой строки включительно и бакет после этой строки исключительно, в каждом из них ищется дисперсия элементов, полученные значения складываются для каждой строки. После этого ищется строка, дающая наименьшую дисперсию при разбиении, и в качестве порогового значения берётся среднее значение элементов с индексом j полученной и следующей строки. Причём если идёт подряд несколько одинаковых значений в заданном столбце, то сравнивается только наиболее позднее из них, так как это реальное значение при разбиении по столбцу j.

Среди наборов оптимальных разбиений по индексу ищется лучший, и этот набор, а также соответствующие индекс и порог разделения, фиксируется и отправляется на следующий этап разделения.

IV. Построение прототипов

Прототипы подбираются таким образом, чтобы тренировочная выборка могла быть восстановлена с как можно меньшим среднеквадратичным отклонением. Это улучшает результаты, так как при этом теряется меньше информации о тренировочной матрице.

Пусть K – число прототипов на подпространство, C – количество самих подпространств, $\left\{J^{(c)}\right\}_{c=1}^{C}$ – разбиение множества индексов, причём каждый блок соответствует своему подпространству, $\boldsymbol{P} \in \mathbb{R}^{KC \times D}$ – матрица с блоками на диагонали размера $K \times |J^{(c)}|$ каждый. Эти блоки содержат по K прототипов в каждом подпространстве c. Тренировочная матрица восстанавливается через выражение $\widehat{\boldsymbol{A}} \approx \boldsymbol{GP}$, где $\boldsymbol{G}$ – матрица выбора соответствующих прототипов в каждом подпространстве. Матрица $\boldsymbol{P}$ строится следующим образом:

$$\boldsymbol{P} = (\boldsymbol{G}^T\boldsymbol{G} + \lambda \boldsymbol{E})^{-1}\boldsymbol{G}^T\widehat{\boldsymbol{A}}$$

Коэффициент λ подбирается с помощью кросс-валидации, но для упрощения алгоритма он фиксируется равным 1.

Подобное построение делает прототипы ненулевыми за пределами их подпространств. Благодаря этому избегаются резко возрастающие накладные расходы, с которыми сталкиваются другие методы с неортогональными прототипами.

V. Таблица поиска

Используя ранее созданные прототипы, алгоритм MADDNESS создаёт таблицы поиска $h^{(c)}(\boldsymbol{b}) \in \mathbb{R}^K$ в каждом из подпространств для каждой колонки матрицы $\boldsymbol{B}$, где

$$h^{(c)}(\boldsymbol{b})_k = \sum_{j \in J^{(c)}} b_j P_{k+(c-1)K,j}.$$

Для уменьшения размера таблицы в памяти все полученные элементы могут быть нормированы так, чтобы полностью использовать требуемую размерность памяти, а обратные преобразования при этом должны быть заложены в константы α и $\boldsymbol{\beta}$.

VI. Агрегация

Пусть $\boldsymbol{T} \in \mathbb{R}^{M \times C \times K}$ – тензор таблиц поиска для всех M столбцов матрицы $\boldsymbol{B}$. Тогда функция $f(\cdot,\cdot)$ определяется как

$$f\big(g(\boldsymbol{A}), h(\boldsymbol{B})\big)_{n,m} = \sum_{c=1}^{C} \boldsymbol{T}_{m,c,k}\,, k = g^{(c)}(\boldsymbol{a}_n).$$

Так как нас интересует в первую очередь аппаратная реализация, то возникает вопрос разрядности хранимых предвычисленных результатов и выходных данных. При каждом суммировании разрядность результата увеличивается на 1 при сохранении точности. Есть следующие варианты аппаратно получить на выходе необходимую разрядность:

1. Снизить разрядность хранимых результатов (проблема – большая начальная ошибка округления)
2. Обрезать младшие биты на выходе последнего сумматора (проблема – расход ресурсов на идущие «в никуда» данные)
3. Использовать операцию попарного усреднения вместо суммирования (компромисс)

Последний вариант кажется на первый взгляд не вполне уместным, но если учесть, что мы нормировали результат коэффициентами α и $\boldsymbol{\beta}$, весь накопленный множитель опять же может быть компенсирован коэффициентом α. Более того, так как в аппаратной реализации нейронных сетей выгоднее применять в плане скорости без больших потерь в точности арифметику с фиксированной точкой вместо арифметики с плавающей точкой [7], а для собственно нейронных сетей важнее относительные значения, а не абсолютные, то можно в аппаратной реализации модуля перейти к целочисленной арифметике и нормировать соответствующим образом всю нейронную сеть. Именно поэтому в аппаратной реализации для нейронных сетей третий вариант наиболее пригоден.

VII. Аппаратная реализация MADDNESS

Для аппаратной реализации MADDNESS был создан шаблон, на основе которого можно сделать реализацию с требуемыми параметрами. Были взяты следующие настраиваемые параметры:

1. Число слоёв классификатора (n)
2. Число деревьев-классификаторов (C)
3. Длина входной строки (D)
4. Длина выходной строки (M)
5. Разрядность модуля

На вход модуля подаётся строка $\boldsymbol{a}$ и тактовый сигнал, а также сигналы, управляющие внутренним модулем памяти (запись, данные и адрес), который хранит все изменяемые константы (номера элементов, которые сравниваются с пороговыми значениями, пороговые значения и частичные матричные произведения). На выходе получается строка, которая является предполагаемым результатом умножения строки $\boldsymbol{a}$ на матрицу $\boldsymbol{B}$ (а именно точное произведение $\widehat{\boldsymbol{a}}$ на $\boldsymbol{B}$) с поправкой на константы α и $\boldsymbol{\beta}$.

Так как модуль должен обрабатывать поток данных, то разрабатывалась именно конвейерная реализация модуля. На рисунке 3 изображён результат синтеза модуля с n = 4 и C = 4. При построении архитектуры модуля в первую очередь конвейер был разбит на 2 этапа, отвечающих за последние этапы алгоритма (связанные непосредственно с обработкой данных, классификатор и сумматор). Те в свою очередь были разбиты на отдельные этапы, не имеющие обратной связи. Классификатор был разбит на отдельные слои, сумматор – на этапы попарного суммирования (по факту – усреднения). Также отдельно был выделен модуль памяти, хранящий индексы $j^1,\ldots,j^n$, наборы эталонов $\boldsymbol{v}^1,\ldots,\boldsymbol{v}^n$ и тензор таблиц поиска $\boldsymbol{T}$.

Модуль MADDNESS построен таким образом,

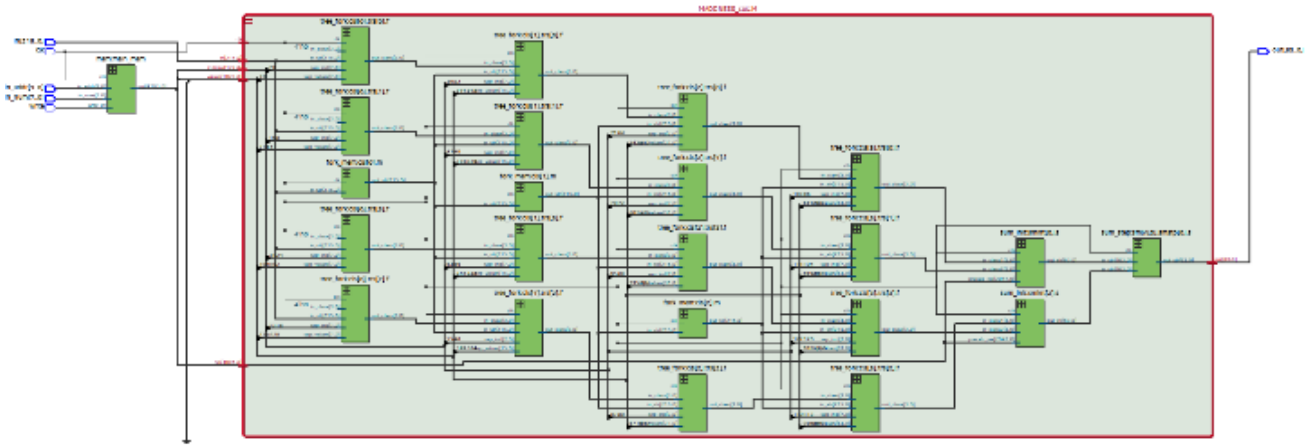

Рис. 1. Результат синтеза модуля MADDNESS

чтобы каждый из подмодулей совершал свою работу ровно за 1 такт. Модули, обрабатывающие один пакет данных в один и тот же такт, расположены на одной вертикали, которую будем называть конвейерным блоком. Для наглядного изображения структуры модуля был проведён синтез средствами Quartus.

На рисунке можно чётко выделить 2 группы модулей: деревья решений (4 левых конвейерных блока) и сумматоры (2 правых конвейерных блока). В группе деревьев решений выделяется 5 конвейерных линий: 4 линии непосредственно деревьев решений и линия, передающая входную строку по этапам конвейера. Группа сумматоров имеет древовидную структуру, так как с каждым этапом сумматоры уменьшают число линий конвейера.

Собственно вычислительный модуль состоит из следующих модулей:

1. tree_fork – классификатор, уровень конкретного дерева решений
2. fork_mem – память конвейера, передающая анализируемую строку
3. sum_init – подстановка промежуточных решений и первое попарное суммирование
4. sum_step – последующие попарные суммирования

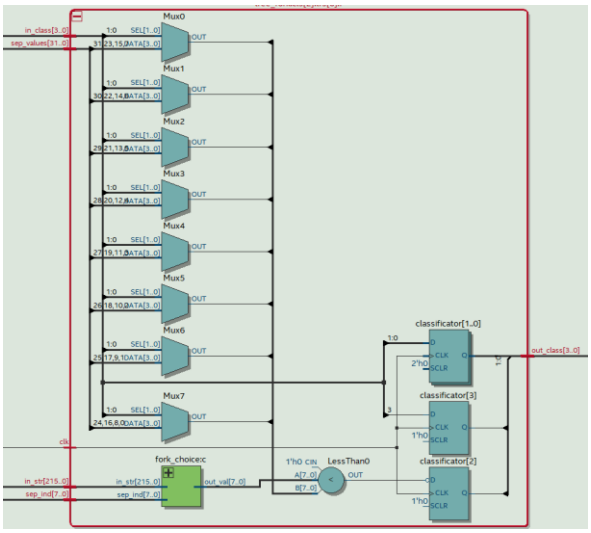

Рис. 4. Результат синтеза tree_fork

VIII. Слой классификатора

На рисунке 4 представлена структура модуля tree_fork, который является этапом классификации отдельного дерева. Сам по себе модуль содержит в себе следующие части:

1. Блок мультиплексоров, выбирающих v_i^t на основе предыдущей классификации (слева сверху). Число мультиплексоров соответствует разрядности модуля.
2. Блок мультиплексоров, выбирающих, какой из элементов строки сравнивать с эталоном (слева снизу). Для удобства отображения скрыты единым модулем.
3. Компаратор, сравнивающий выбранный элемент с выбранным эталоном (в центре снизу).
4. Регистр классификатора, в котором хранится результат классификации с предыдущего этапа и записывается результат классификации на текущем этапе (справа снизу). Так как запись идёт в один из триггеров посредине регистра, на представлении регистр разбит на 3 части.

Параллельно модулям tree_fork расположен модуль fork_mem, хранящий в себе обрабатываемую в данный момент входную строку.

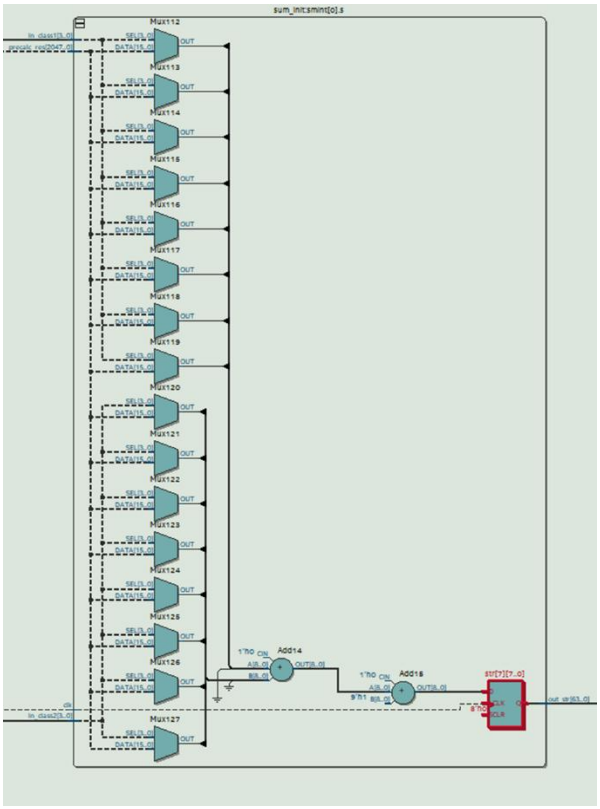

Рис. 5. Результат синтеза модуля sum_init

IX. Слои сумматоров

На рисунке 5 представлена частичная структура модуля sum_init. Данный модуль находится в первом конвейерном блоке сумматоров, так как именно в первом ряду сумматоров производится выбор конкретного частичного скалярного произведения. Сам модуль содержит в себе следующие части:

1. 2 блока мультиплексоров, выбирающих решение исходя из решений классификаторов (слева)

2. 2 сумматора, последовательно складывающих ответы сначала друг с другом, а затем с единицей (в центре снизу)
3. Регистр, хранящий в себе частичную сумму для конкретной компоненты выходной строки

Модуль sum_step является обрезанным вариантом модуля sum_init. В нём отсутствуют мультиплексоры, поскольку на последующих этапах суммирования не требуется обращаться к результатам классификации.

X. Занимаемая площадь и временные ограничения

Помимо средств Quartus, был произведён синтез модуля MADDNESS средствами Cadence Genus с различным числом и размером деревьев. Следующие параметры были зафиксированы:

1. Длина входной строки – 27 элементов
2. Длина выходной строки – 1 элемент
3. Разрядность – 8 бит

В качестве основы была взята библиотека NangateOpenCellLibrary. Также для оценки скорости были синтезированы накопительный матричный умножитель и комбинационный. Результаты показаны на рисунках 6 и 7.

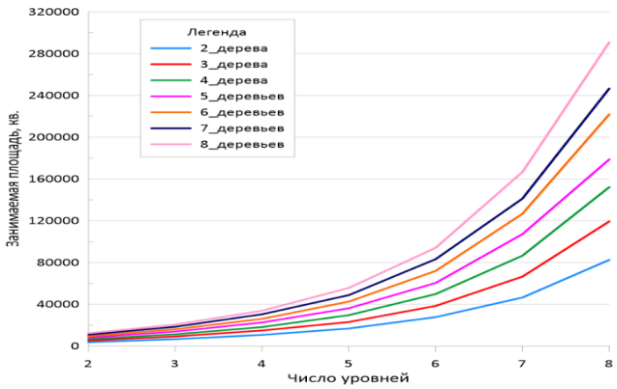

Рис. 2. Зависимость занимаемой модулем площади от глубины деревьев при их разном количестве

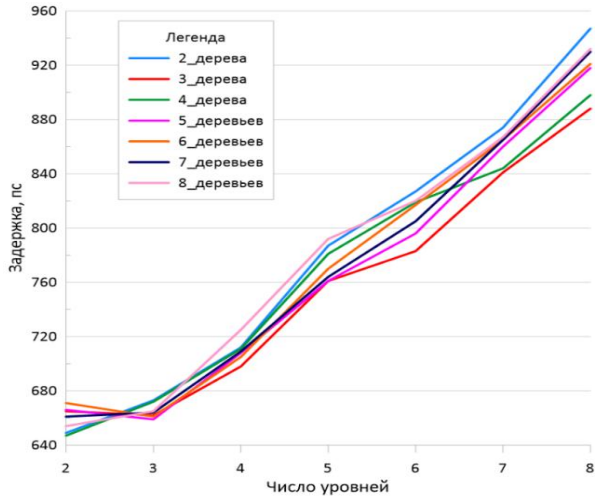

Рис. 3. Зависимость минимальной длины такта от глубины деревьев при их разном количестве

Как видно из графиков, при увеличении глубины деревьев классификации площадь модуля растёт экспоненциально, а размер такта – практически линейно. При этом, если посмотреть критический путь для каждого из модулей, то выяснится, что при глубине дерева 2 ограничивающим фактором будет передача индекса для сравнения из памяти в классификатор, а для прочих – подстановка ответа и первое суммирование в модуле sum_init. Для сравнения: для накопительного умножителя занимаемая площадь 3997.182 квадрата, размер такта 1582 пс. То есть при большей занимаемой площади умножитель из 2 деревьев глубины 8 каждое будет в 1.6 раз быстрее на 1 такт и в 5.6 раз быстрее при обработке одиночных строк, чем накопительный умножитель, при этом возможные ответы покрываются практически полностью при достаточно большом дроблении строки.

XI. Заключение

Алгоритм MADDNESS при относительно высокой точности показывает хорошие результаты по ускорению матричного умножения. При этом простота алгоритма позволяет не использовать умножители, которые сильнее всего тормозят работу схемы. Аппаратная реализация алгоритма позволяет использовать его в качестве составной части более сложного автономного устройства как при реализации в ПЛИС, так и при реализации в виде специализированной интегральной схемы (ASIC). Также полученная реализация позволяет загружать веса в схему «на лету», что позволяет использовать модуль для умножения любых матриц заданных размеров, для которых получены пороги сравнения и предрассчитанные произведения.

Данная реализация имеет потенциал к дальнейшему ускорению, так, можно разделить подстановку решения и первое суммирование решений на отдельные этапы, что ускорит общую работу модуля за счёт уменьшения самого длинного такта.

Проект выполнен при финансовой поддержке гранта РНФ № 22-29-00762.

Литература

[1] J. Mairal, F. Bach, J. Ponce, G. Sapiro. Online dictionary learning for sparse coding // Proceedings of the 26th annual international conference on machine learning, 2009, с. 689–696.
[2] N. Ailon, B. Chazelle. The Fast Johnson–Lindenstrauss Transform and Approximate Nearest Neighbors // SIAM Journal on Computing (SICOMP), 2009, с. 302–322.
[3] A. Dasgupta, R. Kumar, T. Sarlós. A Sparse Johnson–Lindenstrauss Transform // Proceedings of the fortysecond ACM symposium on Theory of computing, 2010, с. 341–350.
[4] H. Jegou, M. Douze, C. Schmid. Product quantization for nearest neighbor search // IEEE transactions on pattern analysis and machine intelligence, 2011, с. 117–128.
[5] D. Blalock, J. Guttag. Bolt: Accelerated Data Mining with Fast Vector Compression // Proceedings of the 23rd ACM

SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, c. 727–735.
[6] D. Blalock, J. Guttag. Multiplying Matrices Without Multiplying // Proceedings of the 38 th International Conference on Machine Learning, 2021, c. 992-1004.
[7] R. Solovyev, A. Kustov, D. Telpukhov, V. Rukhlov, A. Kalinin. Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA // 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 2019, c. 1605-1611.

Hardware Implementation of an Accelerated Approximated Matrix Multiplier Based on MADDNESS Algorithm

A. S. Zhigulin, R. A. Soloviev

Institute for Design Problems in Microelectronics of RAS, Moscow, lenshtein-z@yandex.ru

***Abstract* – The article proposes a technique for creating of hardware implementation of an accelerated approximated matrix multiplier MADDNESS. This multiplier has good performance in terms of accuracy and speed and at same time is distinguished by simplicity of the decoder. As a result of the research, it was possible to achieve a very high speed of the multiplier at the hardware level by rejection of the multiplication operation as such. At the same time, the quality of the obtained prediction remains high.**

***Keywords* – Hardware implementation of matrix multiplying, FPGA, integer arithmetic.**

REFERENCES

[1] J. Mairal, F. Bach, J. Ponce, G. Sapiro. Online dictionary learning for sparse coding // Proceedings of the 26th annual international conference on machine learning, 2009, pp. 689–696.
[2] N. Ailon, B. Chazelle. The Fast Johnson–Lindenstrauss Transform and Approximate Nearest Neighbors // SIAM Journal on Computing (SICOMP), 2009, pp. 302–322.
[3] A. Dasgupta, R. Kumar, T. Sarlós. A Sparse Johnson–Lindenstrauss Transform // Proceedings of the fortysecond ACM symposium on Theory of computing, 2010, pp. 341–350.
[4] H. Jegou, M. Douze, C. Schmid. Product quantization for nearest neighbor search // IEEE transactions on pattern analysis and machine intelligence, 2011, pp. 117–128.
[5] D. Blalock, J. Guttag. Bolt: Accelerated Data Mining with Fast Vector Compression // Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 727–735.
[6] D. Blalock, J. Guttag. Multiplying Matrices Without Multiplying // Proceedings of the 38 th International Conference on Machine Learning, 2021, pp. 992-1004.
[7] R. Solovyev, A. Kustov, D. Telpukhov, V. Rukhlov, A. Kalinin. Fixed-Point Convolutional Neural Network for Real-Time Video Processing in FPGA // 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 2019, pp. 1605-1611.