OM FEST 2025

ФЕСТИВАЛЬ ЦИФРОВЫХ ТЕХНОЛОГИЙ

Как писать быстрый код под Эльбрус

Мария Богданова
НИЦ ЦТ, разработчик

Содержание:

Предпосылки:

Когда речь идет о нестандартных или специализированных архитектурах и процессорах, то **общепринятые методы оптимизации далеко не всегда дают желаемый результат.**

Также, немаловажной причиной является отсутствие готовых библиотек у нестандартных архитектур.

Это и является одной из предпосылок для более глубокого изучения специфики, такой как внутренней работы процессора, особенностей его конвейера, кэш-памяти, а также специфических наборов инструкций.

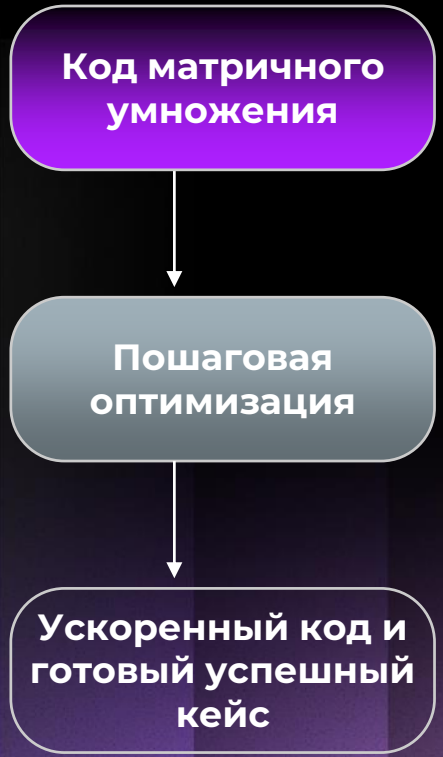
Так и появилась цель: проанализировать и исследовать архитектурные нюансы вручную для последующей адаптации и пошаговой оптимизации кода

Цель:

Насколько можно ускорить один и тот же код, если целенаправленно применять приёмы оптимизации именно под e2k?

Подопытный кролик:
тест матричного умножения.

Тест представляет собой реализацию типичной задачи линейной алгебры, а именно перемножения двух квадратных матриц.



Что узнаем?

1. время, необходимое на перемножение
2. оценку вычислительной мощности процессора
3. эффективность реализации математических операций
4. узкие места в коде и возможности для оптимизации (разбиение на блоки, векторизация, развертывание циклов, предварительная выборка данных)

Особенности Эльбрус

VLIW (Very Long Instruction Word):

Особенности:

1. Процессоры «Эльбрус» способны выполнять до 25 инструкций за такт.
2. Явный параллелизм и статическое планирование.
3. Компилятор, а не процессор, отвечает за порядок выполнения команд.
4. Компилятор должен эффективно "упаковывать" инструкции в длинные машинные команды.

Неоптимальное планирование инструкций может привести к **простоям исполнительных устройств и существенному снижению производительности.**

Большой регистровый пул:

1. 256 регистров, которые доступны для всех функций.
2. Возможно агрессивное разворачивание циклов
3. Возможна реализация вычислений на регистрах без промежуточных обращений в память.

SIMD-инструкции:

Начиная с 5-й версии архитектуры Эльбрус, процессоры поддерживают векторные SIMD-инструкции, работающие со 128-битными векторами.

Эти инструкции позволяют повысить эффективность обработки данных, но требуют специфической адаптации кода.

Компилятор предоставляет программисту интринсики, эмулирующие SSE и AVX инструкции архитектуры x86_64, но не все из них реализованы аппаратно, что необходимо учитывать при написании кода.

Подготовка окружения

Выбор дистрибутива:

Для корректного воспроизведения и работы над тестом, важно иметь доступ к дистрибутиву, где установлены необходимые компоненты, например - библиотека EML.

В случае отсутствия данной либы, нужно **заранее настроить окружение.**

1. Загрузка EML
1. Установка EML

Запуск на 2c3:

Запуск и работа над тестом происходила на процессоре E2C3(архитектура Эльбрус v6), двумя ядрами и оперативной памятью в размере 16 ГБ.

Для чистоты эксперимента, необходимо изолировать ядро под выполнение тестов, и благодаря изоляции, будет оцениваться производительность конкретно для запущенного теста умножения матриц, без каких-либо “лишних” процессов.

Для изоляции ядра, необходимо изменить параметры в boot.conf (в (/boot/boot.conf) добавить для cmd команды: isolcpus=1)

Методики оптимизаций

Отправная точка:

Компиляция:
/$ lcc -O3 -fwrapv -flto -o matmul matmul.c

Запуск и замер времени:
/$ time taskset -c 1 ./matmul

Результат:
0m46.076s

-fwrapv - флажок, который заставляет компилятор считать, что при переполнении знаковых чисел (signed integer overflow) происходит “оборачивание” по модулю (wrap around).

-flto - включает Link Time Optimization (оптимизацию на этапе линковки).
Позволяет компилятору оптимизировать весь проект целиком, улучшая производительность итогового кода.

taskset -c 1 - запускает программу на конкретном ядре CPU с номером 1. Это обеспечивает изоляцию процесса на одно ядро, что улучшает воспроизводимость замеров производительности

Профилирование тривиального решения:

Профилирование осуществлялось с помощью perf:
perf record ./matmul
perf report

В данном коде, вся активность и нагрузка находится в функции main, так как в ней вызывается тяжёлая вычислительная функция mat_mul - об этом явно говорит результат профилирования.

Overhead (%)	Command	Shared Object	Symbol	Описание
100.00%	matmul	matmul	.main	Вся нагрузка сосредоточена в функции main, которая содержит вызов mat_mul.
...	...	...	...	...

А зачем это всё? Какой “максимум” может выдать Эльбрус?

Код с использованием EML:

В качестве целевой точки исследования “а что может показать Эльбрус”, взят результат оптимизации, достигнутый с использованием специализированной библиотеки EML (Эльбрус Математическая Библиотека), содержащей оптимизированные реализации функций, в том числе функции линейной алгебры.

Результат:

```
/$ time taskset -c 1 ./matmul-eml
real    0m0.203s
```

Данные показатели заставили задуматься: можно ли добиться подобной производительности собственными силами, без опоры на сторонние библиотеки?
Какие весомые различия есть между кодами?

С EML и без

Работа с памятью:

```
double *mat_alloc(int n_row, int n_col)
{
        double *mat, *a;
        int i;
        a = (double*)calloc(n_row * n_col,
sizeof(double)); // 1D-массив для хранения
элементов
        mat = (double*)malloc(n_row *
sizeof(void*));        // массив указателей на
строки
        for (i = 0; i < n_row; ++i)
        mat[i] = &a[i * n_col];       // задаём
указатели на начало каждой строки
        return mat;
}
```

Базовый код теста matmul

```
int i, j, N = 1500;

double *A = (double *)malloc(N * N *
sizeof(double));
double *B = (double *)malloc(N * N *
sizeof(double));
double *C = (double *)malloc(N * N *
sizeof(double));
```

Код теста matmul с EML

Заполнение матриц:

```
double **mat_gen(int n)
{
        double **a, tmp = 1.0 / n / n;
        int i, j;
        a = mat_alloc(n, n);
        for (i = 0; i < n; ++i)
                for (j = 0; j < n; ++j)
                        a[i][j] = tmp * (i
- j) * (i + j);
        return a;
}
```

Базовый код теста matmul

```
double tmp = 1.0 / (N * N);
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i * N + j] = tmp * (i - j) * (i + j);
    B[i * N + j] = tmp * (i - j) * (i + j);
  }
}
```

Код теста matmul с EML

Перемножение матриц:

*C = A * B*

```
double **mat_mul(int n, int p, double **a, int
m, double **b)
{
        double **c;
        int i, j, k;
        c = mat_alloc(n, m);
        for (i = 0; i < n; ++i)
                        for (k = 0; k < p; ++k)
                                        for (j = 0; j < m;
++j)

        c[i][j] += a[i][k] * b[k][j];
        return c;
}
```

Базовый код теста matmul

*C = ALPHA * A * B + BETA * C*

```
eml_Algebra_GEMM_64F(
    EML_MATRIX_ROW_MAJOR,
    EML_MATRIX_NO_TRANS,
    EML_MATRIX_NO_TRANS,
    N, N, N,
    1,      // коэффициент масштабирования
ALPHA
    A, N,
    B, N,
    0,      // коэффициент масштабирования
BETA
    C, N
);
```

Код теста matmul с EML

Сравнение времени:

0m46.076s

Базовый код теста matmul

0m0.203s

Код теста matmul с EML

Составление алгоритма оптимизации без EML

Анализ “направлений” оптимизации:

1. Формат хранения
2. Одномерное индексирование матриц
3. Блочный алгоритм и автовекторизация
4. Распараллеливание

Формат хранения:

Можно оптимизировать выделение и хранение памяти под матрицы.

Выделение множества разрозненных блоков для каждой строки

Сохранение всех данных в одном непрерывном блоке памяти

Переход к одномерности:

Можно перейти к одномерности и упростить индексацию.

Использование двумерных массивов

Использование одномерных массивов

Блочный алгоритм:

Обработка блоками - один из возможных подходов.

Стандартная обработка массивов

Блочная обработка массивов,
или же “блочное перемножение матриц”

Этапы оптимизации и их результаты

Задача - постепенно улучшать базовый код, приближаясь к показателям кода с EML

Шаг 1.
Выравнивание памяти

Путешествие в тысячу ли
начинается с первого шага

Первый шаг - локализация памяти:

```
static double **mat_alloc(int n_row, int n_col)
/* возвращает квадратную матрицу n_row×n_col, размещённую одним
блоком  */
{ size_t hdr  = (size_t)n_row * sizeof(double *);
   size_t body = (size_t)n_row * n_col * sizeof(double);
 char *block = (char *)calloc(1, hdr + body);
   if (!block) return NULL;
double **row = (double **)block;
   double  *dat = (double *)(block + hdr);
 for (int i = 0; i < n_row; ++i)
       row[i] = dat + (size_t)i * n_col;
return row;  }
```

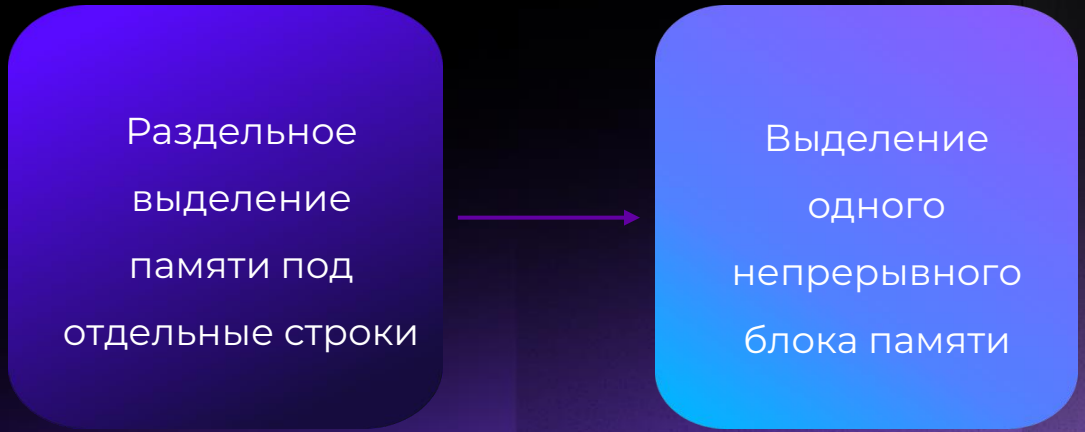
Первый шаг - локализация памяти:

- Основное отличие оптимизированного кода, полученного в результате первого шага от оригинального теста matmul заключается в подходе к выделению памяти под матрицы.

- Вместо традиционного раздельного выделения памяти под отдельные строки, теперь используется функция, которая выделяет всю память одним непрерывным блоком.

Результаты первого шага:

1. Уменьшение фрагментации
2. Улучшение локальности данных
3. Упрощение освобождения памяти



Результаты первого шага:

/$ time taskset -c 1 ./matmul-opt-line-mem

0m45.963s

Изменение на уровне ~ 0.2%, которое хорошо себя проявит в дальнейшем.

Сравнение времени:

0m46.076s

Базовый код теста matmul

0m45.963s

Код теста matmul после выравнивания памяти

Шаг 2.
Переход к одномерности

Второй шаг - одномерность:

Функция выделения памяти:

```
double *A = malloc((size_t)N * N * sizeof *A);
double *B = malloc((size_t)N * N * sizeof *B);
double *C = calloc((size_t)N * N, sizeof *C);
```

Второй шаг - одномерность:

Функция перемножения матриц:

```
for (int i = 0; i < N; ++i)
    for (int k = 0; k < N; ++k) {
        for (int j = 0; j < N; ++j)
            C[i * N + j] += A[i * N + k]; * B[k * N + j];
    }
```

Результаты второго шага:

~/line-matrix$ time taskset -c 1 ./matmul-opt-line

0m2.307s

В этом шаге происходит упрощение структуры данных, за счет сведения двумерных массивы к одномерным.

Одномерный массив позволяет обрабатывать элементы последовательно и предсказуемо, что облегчает кэширование и минимизирует обращения к памяти.

Сравнение времени:

0m45.963s

Код теста matmul после выравнивания памяти

0m2.307s

Код теста matmul после перехода к одномерности

Шаг 3.
Реализация блочного умножения

Третий шаг - реализация блочного умножения:

1. Матрица C разбивается на квадратные микро-плитки MICRO × MICRO. Внешние циклы i и j перебирают эти плитки по строкам и столбцам.

2. Для каждой выходной плитки создаётся нулевой аккумулятор

3. Цикл k последовательно «пробегает» общую размерность и строит ОДИН «малый» столбец из A (av[]) и одну «малую» строку из B (bv[]).

Третий шаг - реализация блочного умножения:

4. Двойной внутренний цикл di,dj считает их наружное произведение av * bv и прибавляет к аккумулятору acc.

5. После того как все k перебраны, содержимое acc записывается обратно в соответствующую плитку C.

6. Переходим к следующему столбцу плиток (j += MICRO), потом к следующей строке плиток (i += MICRO) — пока не обработаем всю матрицу.

Результаты третьего шага:

../matmul/tile$ time taskset -c 1 ./matmul-opt-tile

0m0.528s

Сравнение времени:

0m2.307s

Код теста matmul после перехода к одномерности

0m0.528s

Код теста matmul после реализации блочного умножения

Вывод:

Оценив показатели до и после, можно смело сказать, что удалось значительно ускорить код перемножения матриц, при этом не использовав специфичные библиотеки.

Базовый код:	После шага №1:	После шага №2:	После шага №3:	Магия EML:
0m46.076s	**0m45.963s**	**0m2.307s**	**0m0.528s**	**0m0.203s**

“Пиши код аккуратно и осмысленно — и хороший результат не заставит себя ждать!”

КОНТАКТЫ

Сайт компании НИЦ ЦТ

Наши статьи

Моя почта для вопросов и обсуждения исследования:
m.bogdanova@nicct.ru

OM FEST 2025

ФЕСТИВАЛЬ ЦИФРОВЫХ ТЕХНОЛОГИЙ

Как писать быстрый код под Эльбрус

Время для вопросов!

Мария Богданова
НИЦ ЦТ, разработчик