Проблемы масштабируемости, связанные с архитектурой: особенности буфера ассоциативной трансляции - TLB.

Подготовил Павел Никишкин
323 группа, СКИ ВМК

Московский государственный университет имени М. В. Ломоносова, факультет ВМК

Проблема

TLB, или Translation Lookaside Buffer, это кэш-память, используемая процессором для хранения часто используемых адресов виртуальной памяти и их соответствующих физических адресов. Проблемы масштабируемости, связанные с TLB, могут возникать при работе с большими объемами данных или при многопоточной обработке. Вот некоторые из них:

1. TLB промахи
2. Конкуренция за ресурсы
3. Обновление TLB (при изменении таблиц страниц или при изменении контекста, например, при переключении процессов)
4. Фрагментация TLB

Пример – параллельная обработка вектора

Рассмотрим алгоритм поиска суммы элементов вектора. В качестве не оптимизированного примера разделим вектор на несколько нитей следующим образом: каждая нить вычисляет сумму элементов, которые находятся через смещение друг от друга. Смещение при этом равно общему числу нитей. При этом при увеличении числе нитей возможно возникновение проблем с производительностью из-за кэш-промахов, так как увеличение числа нитей приводит к увеличению смещения, из-за чего происходит обращение к блокам данных, которые находятся далеко друг от друга в памяти.

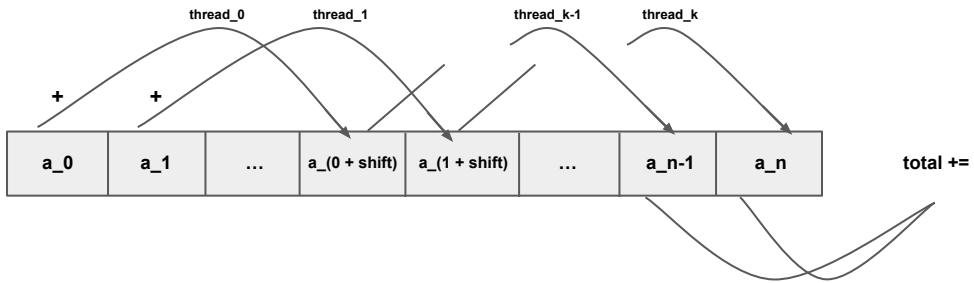

Иллюстрация алгоритма

Пример – параллельная обработка вектора

При этом есть возможность разделить обрабатываемый вектор на блоки по нитям и обрабатывать их отдельно друг от друга. Таким образом, мы можем изменить распределение блоков данных так, чтобы они находились ближе друг к другу в памяти, чтобы уменьшить обращения к разным страницам виртуальной памяти, тем самым уменьшить TLB промахи.

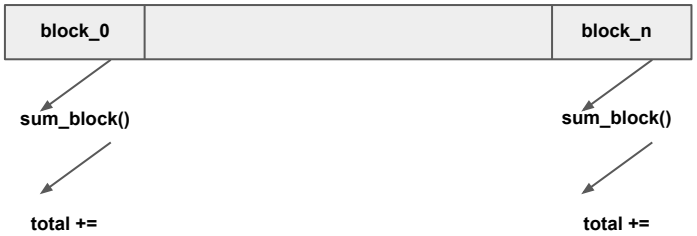

Иллюстрация алгоритма с разделением на блоки

Сравнение версий

Проблемная версия

```
// Функция для параллельного вычисления суммы элементов вектора
void parallel_accumulate(int start, int shift_size, int size, std::vector<double> a, double& total) {
    double res = 0;
    for (int i = start; i < size; i += shift_size) {
        res += a[i];
    }
    total += res;
}
...
// Разделение работы между потоками OpenMP
    #pragma omp parallel shared(total)
    {
        int thread_id = omp_get_thread_num();
        int shift_size = num_threads;
        int start_index = thread_id;
        parallel_accumulate(start_index, shift_size, size, data, total);
    }
```

Оптимизированная версия

```
// Функция для параллельного вычисления суммы элементов вектора
void parallel_accumulate(int start, int end, std::vector<double> a, double& total) {
    double res = 0;
    for (int i = start; i <= end; ++i) {
        res += a[i];
    }
    total += res;
}
...
// Разделение работы между потоками OpenMP
    #pragma omp parallel shared(total)
    {
        int thread_id = omp_get_thread_num();
        int block_size = size / num_threads;
        int start_index = thread_id * block_size;
        int end_index = (thread_id == num_threads - 1) ? size : start_index + block_size;
        parallel_accumulate(start_index, end_index, size, data, total);
    }
```

Параметры тестирования

1. массив double из 10^9 случайных элементов (нормальное распределение на отрезке [-1.0, 1.0]).
2. Параметры системы:

   Polus - параллельная вычислительная система, 5 вычислительных узлов.

   Основные характеристики каждого узла:

   - 2 десятиядерных процессора IBM POWER8 (каждое ядро имеет 8 потоков) всего 160 потоков
   - Общая оперативная память 256 Гбайт (в узле 5 оперативная память 1024 Гбайт) с ECC контролем
   - 2 x 1 ТБ 2.5” 7K RPM SATA HDD
   - 2 x NVIDIA Tesla P100 GPU, 16Gb, NVLink
   - 1 порт 100 ГБ/сек

Результаты тестирования

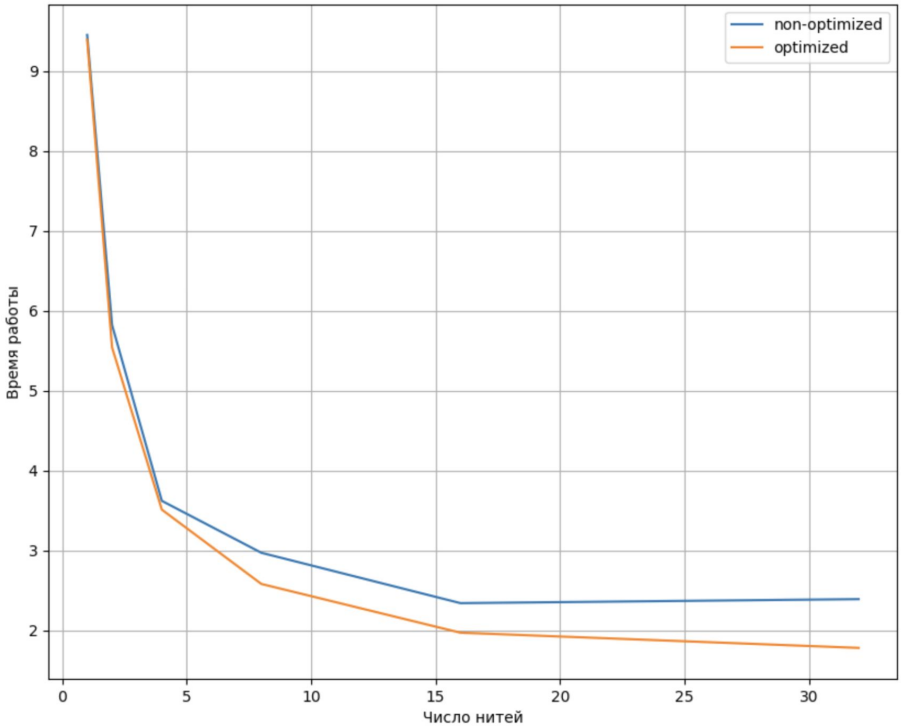

График зависимости времени выполнения от числа нитей

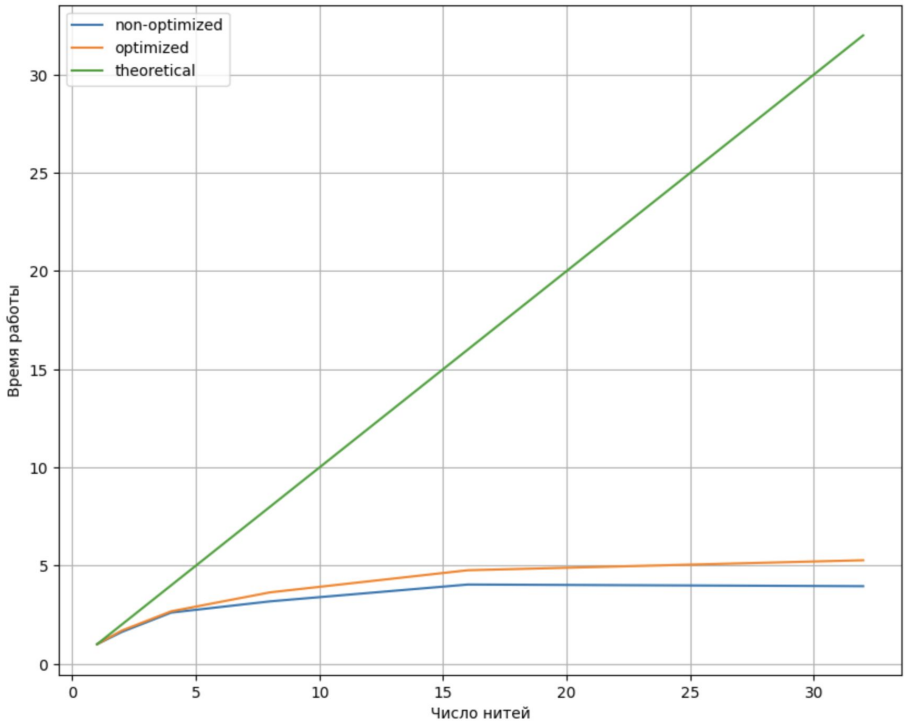

График зависимости ускорения от числа нитей