PJ4 优化

Your classmate posted a new Note.

PJ4 优化

PJ4 优化

PJ4是关于Iterative Stencil Loops中2D Jacobi iteration的计算。从题设的评判标准看，迭代的正确性检验，是通过取你的结果与倒数第一步与倒数第二步差值的最小值平方，再累加与EPS=1e-5相比较。

本题的题设是，给定大小为N的矩阵，应用2D Jacobi iteration计算矩阵中非边缘元素的k步迭代。注意到2D Jacobi iteration的计算模式，是取矩阵非边缘位置的相邻元素算平均值。

我们的第一步优化可以是：

- 奇数步：从第0行第0列开始计数，在奇数行m计算第m行第${1, 3, 5, 7, ...}$列的值，在偶数行n计算第n行第${2, 4, 6, 8, ...}$列的值。跳过第0行、第$N-1$行、第0列，第$N-1$列；
- 偶数步：从第0行第0列开始计数，在奇数行m计算第m行第${2, 4, 6, 8, ...}$列的值，在偶数行n计算第n行第${1, 3, 5, 7, ...}$列的值。跳过第0行、第$N-1$行、第0列，第$N-1$列。

正确性留待读者自证。

在结束了算法上的优化后，注意到我们每次以间隔列元素的方式进行计算，不方便使用SIMD优化，cache访问也不友好。因此我们将矩阵进行重排。

- 首先，考虑使用`aligned_alloc(32, <size>)`存放重排后的矩阵。这个函数可以在分配内存时按第一个参数规定的大小进行内存对齐，从而减少访存开销。函数的具体用法见cppreference。
- 其次，考虑到我们的计算方法，我们使用两组矩阵A, B，分别存放奇数步的计算结果与偶数步的计算结果。设原矩阵为P，则A, B与原矩阵的对应关系是： $$ A = {a_{i, \lfloor\frac{j}{2}\rfloor}: a_{i, \lfloor\frac{j}{2}\rfloor} = P_{i, j}, i + j\ (\text{mod}\ 2) \equiv 1, i, j \in [0, N) \cap \mathbb{Z} } $$ $$ B = {b_{i, \lfloor\frac{j}{2}\rfloor}: b_{i, \lfloor\frac{j}{2}\rfloor} = P_{i, j}, i + j\ (\text{mod}\ 2) \equiv 0, i, j \in [0, N) \cap \mathbb{Z} } $$

在矩阵的重排过程中，注意到当N分别为奇数和偶数时，A和B每行存放的列元素数量有所区别。具体来说，以A为例。

- 当N为偶数时，A每行存放的列元素均为$N/2$个；
- 当N为奇数时，A的奇数行存放$\lfloor\frac{N}{2}\rfloor + 1$个元素，$A$的偶数行存放$\lfloor\frac{N}{2}\rfloor$个元素。
- B的情况与之类似，请读者自行推导。

虽然N分别为偶数和奇数时，A和B的存放模式不同，我们仍然可以以相同的模式进行计算（见下）。

- 通过以上对应关系，
- 在奇数步时，我们从A矩阵提取数据，进行计算，并将结果存放至B矩阵。计算公式如下： $$ B_{i, j} = (A_{i-1, j} + A_{i, j} + A_{i+1, j} + A_{i, j - (-1)^i}) / 4, i \in [1, N-1), j \in [1 - (i \text{ and } 1), \lfloor\frac{N}{2}\rfloor] $$
- 在偶数步时，我们从B矩阵提取数据，进行计算，并将结果存放至A矩阵。计算公式如下： $$ A_{i, j} = (B_{i-1, j} + B_{i, j} + B_{i+1, j} + B_{i, j + (-1)^i}) / 4, i \in [1, N-1), j \in [i \text{ and } 1, \lfloor\frac{N}{2}\rfloor] $$

由于目前矩阵的访问模式已经变成了按照列元素逐次访问，因此我们可以引入SIMD。

- SIMD计算上述表达式略。注意到AVX指令集最多支持同时计算4个`double`，因此我们选择`_mm256`类指令。需要注意的是，在Intrinsics Guide上，你可以查看到不同SIMD指令的CPI。注意到`div`类指令的CPI为8，远高于`mul`类指令的CPI`0.5`，而编译器通常不会自动将SIMD的`div`指令优化为`mul`指令，因此我们需要手动将`div 4`改为`mul 0.25`。此外，还需要注意使用SIMD时，需要处理不能被`4`除尽的N的情况。这时你需要一些SISD的操作。
- 本项目无需Cache Blocking，原因见Profile结果（后文）。
- 最后，为每个可能加入OpenMP的循环引入OpenMP。
- 样例代码：

```
void impl(int N, int step, double *p) {
double divisor[4] = {
    0.25f,
    0.25f,
    0.25f,
    0.25f,
};
__m256d p_divisor = _mm256_loadu_pd(divisor);

// rearrange
int N2 = (N + 1) / 2;
double *p_part[2] = {
    aligned_alloc(32, N2 * N * sizeof(double)),
    aligned_alloc(32, N2 * N * sizeof(double)),
};
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    int part = i & 1;
    for (int j = 0; j < N; ++j) {
    p_part[part][i * N2 + j / 2] = p[i * N + j];
    part ^= 1;
    }
}

// caculate
int INPUTpartID = 1;
int OUTPUTpartID = 0;

if (N & 1) { // N = odd
    for (int k = 0; k < step; k++) {
#pragma omp parallel for
    for (int i = 1; i < N - 1; i++) {
        int j_head = (INPUTpartID + i) & 1;
        int j_begin = i * N2 + j_head;
        int j_end = (i + 1) * N2 - 1;
        int j = j_begin;
        for (; j < j_end - 3; j += 4) {
        __m256d p1 = _mm256_loadu_pd(&p_part[INPUTpartID][j - N2]);
        __m256d p2 = _mm256_loadu_pd(&p_part[INPUTpartID][j - j_head]);
        __m256d p3 = _mm256_loadu_pd(&p_part[INPUTpartID][1 + j - j_head]);
        __m256d p4 = _mm256_loadu_pd(&p_part[INPUTpartID][j + N2]);
        __m256d sum1 = _mm256_add_pd(p1, p2);
        __m256d sum2 = _mm256_add_pd(p3, p4);
        __m256d sum3 = _mm256_add_pd(sum1, sum2);
        __m256d result = _mm256_mul_pd(sum3, p_divisor);
        _mm256_storeu_pd(&p_part[OUTPUTpartID][j], result);
        }

        // for the tail
        for (; j < j_end; j++) {
        double p1 = p_part[INPUTpartID][j - N2];
        double p2 = p_part[INPUTpartID][j - j_head];
        double p3 = p_part[INPUTpartID][1 + j - j_head];
        double p4 = p_part[INPUTpartID][j + N2];
        p_part[OUTPUTpartID][j] = (p1 + p2 + p3 + p4) / 4.0f;
        }
    }

    int temp = INPUTpartID;
    INPUTpartID = OUTPUTpartID;
    OUTPUTpartID = temp;
    }
} else { // N = even
    for (int k = 0; k < step; k++) {
#pragma omp parallel for
    for (int i = 1; i < N - 1; i++) {
        int j_head = (INPUTpartID + i) & 1;
        int j_begin = i * N2 + j_head;
        int j_end = N2 - 1 + j_begin;
        int j = j_begin;
        for (; j < j_end - 3; j += 4) {
        __m256d p1 = _mm256_loadu_pd(&p_part[INPUTpartID][j - N2]);
        __m256d p2 = _mm256_loadu_pd(&p_part[INPUTpartID][j - j_head]);
        __m256d p3 = _mm256_loadu_pd(&p_part[INPUTpartID][1 + j - j_head]);
        __m256d p4 = _mm256_loadu_pd(&p_part[INPUTpartID][j + N2]);
        __m256d sum1 = _mm256_add_pd(p1, p2);
        __m256d sum2 = _mm256_add_pd(p3, p4);
        __m256d sum3 = _mm256_add_pd(sum1, sum2);
        __m256d result = _mm256_mul_pd(sum3, p_divisor);
        _mm256_storeu_pd(&p_part[OUTPUTpartID][j], result);
        }

        // for the tail
        for (; j < j_end; j++) {
        double p1 = p_part[INPUTpartID][j - N2];
        double p2 = p_part[INPUTpartID][j - j_head];
        double p3 = p_part[INPUTpartID][1 + j - j_head];
        double p4 = p_part[INPUTpartID][j + N2];
        p_part[OUTPUTpartID][j] = (p1 + p2 + p3 + p4) / 4.0f;
        }
    }

    int temp = INPUTpartID;
    INPUTpartID = OUTPUTpartID;
    OUTPUTpartID = temp;
    }
}

// rearrange back
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    int part = i & 1;
    for (int j = 0; j < N; ++j) {
    p[i * N + j] = p_part[part][i * N2 + j / 2];
    part ^= 1;
    }
}

free(p_part[0]);
free(p_part[1]);
}
```

附：Profile（一会更）

Click here to view. Search or link to this question with @514.

Sign up for more classes at http://piazza.com/shanghaitech.edu.cn.

Thanks,
The Piazza Team
--
Contact us at team@piazza.com