并行计算大作业

——并行优化矩阵乘法

朱桐 10175102111
周亦然 10175102207

2019.12.27

«««< HEAD

solving matrix mult on CUDA

- host vs device, distributed memory
- streamprocessors
- blocks, wraps, threads, shared memory

子任务：给定 i, j, 计算 C_{ij}，也就是并行最外面的两个 for

solving matrix mult on CUDA

- 轻量级线程，切换成本较低
- 基本可以平等划分任务
- 一个线程对应一个子任务（开 $n \times m$ 个线程）

shared memory

- 公式: $C_{ij} = \sum_{k=0}^{bm} A_{ik} \cdot B_{kj}$
- 对于内层的两个循环 j, k，发现公用 A_{ix}
- 使用 share memory 缓存 A

share memory

- block 内公用
- `block_size` 必须是 am 的因子，保证同块内使用相同的一行
- 别忘了 `__sychronize`

Fast IO: buffered input & output

- `fread`, `fwrite` instead of `printf`, `scanf`
- no need to flush every time we read/write a number

Occupancy: optimal size for the number of blocks and threads

- 以 wrap 为计算单元
- thread 中有 register
- block 中有 share memory
- 每个 multiprocessor 也有最大的 register 和 share memory 和 active threads
- 如果资源不够将无法派出所有的资源进行运算

Occupancy: optimal size for the number of blocks and threads

- 我们需要让最大比例的显卡工作
- 引入 Occupancy 机制
- 通过 nvidia 官网上的 Occupancy calculator 进行运算
- `nvcc --ptxas-options=-v to nvcc` 查看内核函数使用寄存器个数
- 使用

Bottleneck: IO

- GPU 计算矩阵瓶颈在于 IO
- 输出浮点数也会带来计算

Bottleneck: IO

- 仿照 MPI 在 distributed memory 中使用异步 IO
- Host 一边传入 Device 的数据一边输出到文件
- Host 一边读入文件一遍 Copy 数据到 Device

asychronize IO: input

- `cudaStream`$_t$`cudaHostAlloc`$insteadof$`malloc`
- `cudaMemcpyAsync` instead of `cudaMemcpy`

========

Outline

code: buffered input

```
inline char next_char()
{
    static char buf[100000], *p1 = buf,
        *p2 = buf;
    return p1 == p2 && (p2 = (p1 = buf)
        + fread(buf, 1, 100000, fin), p1 == p2)
        ? EOF : *p1++;
}
```

code: buffered output

```
inline void flush() {
    fwrite(buffer, 1, s-buffer, stdout);
    s = buffer;
    fflush(stdout);
}
inline void print(const char ch) {
    // putchar(ch); return;
    if (s-buffer>OutputBufferSize-2) flush();
    *s++ = ch;
}
```

code: share memory, block partition

```
__shared__ ld c_a[max_shared_size];
int index = blockDim.x * blockIdx.x + threadIdx.x;
if (index >= an * bm) return;
int st = min(index, addi) * (workload+1) +
max(0, index - addi) * workload, ed =
st + workload + (index < addi ? 1 : 0);
int shareda = min(am, max_shared_size);
for (int p=st; p<ed; ++p) {
    // ...
```

code: share memory, block partition

```
for (int p=st; p<ed; ++p) {
    int i = p / bm, j = p % bm;
    if (p % bm == 0) {

        for (int j=0; j<shareda; ++j) {
            c_a[j] = d_a[i * am + j];
        }
        __syncthreads();
    }
    ...
```

code: share memory, block partition

```
for (int p=st; p<ed; ++p) {
    int i = p / bm, j = p % bm;
    if (p % bm == 0) {

        for (int j=0; j<shareda; ++j) {
            c_a[j] = d_a[i * am + j];
        }
        __syncthreads();
    }
    ...
```

code: async IO, creating streams

```
int st = 0, ed = n * m;
// printf("st=%d ed=%d, a=%p\n", st, ed, a);
cudaStream_t stream[2];
int mask = 0;
cudaStreamCreate(&stream[0]);
cudaStreamCreate(&stream[1]);
int size;
```

code: async IO, creating streams

```
cudaStream_t mainstream;
cudaStreamCreate(&mainstream);
// ...
copyMatrixAsync(h_a, d_a, an, am, mainstream);
// ... read h_b
copyMatrixAsync(h_b, d_b, bn, bm, mainstream);
// ...
matrixMult<<<grids, block_size, 0, mainstream>>>
(d_a, d_b, d_c, an, bm, am);
// ...
handleCudaError(cudaStreamSynchronize(mainstream));
// ...
outputMatrixAsync(h_c, d_c, n, m);
// pipeline: memcpy from device & output each rows
```

code: async IO, pipelines

```
for (; st<ed; st+=size, mask^=1) {
    size = min(chunk_size, ed - st);
    handleCudaError(cudaMemcpyAsync(a + st,
     d_a + st, size * sizeof(ld),
     cudaMemcpyDeviceToHost, stream[mask]));
    // exit(0);
    if (st - chunk_size >= 0) {
        // printf("%d %d\n",st-chunk_size, st);
        handleCudaError(cudaStreamSynchronize(strea
        outputinterval(a, st-chunk_size, st);
    }
}
```

Backgrounds

- 1111111111111
- 2222222222222
- 3333333333333
- 4444444444444

My Photo

图: hahahaha...

Emmm...

Sequence Tagging Loss

$$\mathcal{L}_p = -\sum_{i=1}^{S}\sum_{j=1}^{N} p_{i,j}\log(\hat{p}_{i,j})$$

Language Classifier Loss

$$\mathcal{L}_a = -\sum_{i=1}^{S} l_i \log(\hat{l}_i)$$

Bidirectional Language Model Loss

$$\mathcal{L}_l = -\sum_{i=1}^{S}\sum_{j=1}^{N} \log(P(w_{j+1}|f_j)) + \log(P(w_{j-1}|b_j))$$

References

Xuezhe Ma and Eduard Hovy. (2016).
End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF.
In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pages 1064–1074, Berlin, Germany, August 7-12, 2016.

Marek Rei. (2017).
Semi-supervised Multitask Learning for Sequence Labeling.
In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, pages 2121–2130, Vancouver, Canada, July 30 - August 4, 2017.

»»»> 50c2ce376f5f1aca4cb8024ef13d564ad9352d9e