并行计算Lab4实验报告

方驰正PB21000163

一、实验目的

使用cuda编写并行程序，加速fft算法。

二、实验过程

我们分别编写了cpu串行版本和gpu并行版本的fft算法，并对比了两者的性能。

1. 串行版本

```
void FFT(vector<complex<double>> &a, int n, int *rev, int tp) {
    rev[0] = 0;
    for (int i = 1; i < n; i++)
        rev[i] = (rev[i >> 1] >> 1) | ((i & 1) ? n >> 1 : 0);

    for (int i = 0; i < n; i++)
        if (rev[i] > i) swap(a[i], a[rev[i]]);
    for (int l = 1; l < n; l <<= 1) {
        complex<double> w0(cos(M_PI / l), tp * sin(M_PI / l));
        for (int i = 0; i < n; i += l << 1) {
            complex<double> w(1, 0);
            for (int j = i; j < i + l; j++) {
                auto x = a[j], y = a[j + l] * w;
                a[j] = x + y, a[j + l] = x - y, w = w * w0;
            }
        }
    }
}
```

如上所示，我们使用带有位翻转的蝶形运算的fft算法，将递归的算法改为迭代的算法，以减少递归的开销。

2. 并行版本

我们通过编写若干个kernel函数，实现并行化的fft算法。

首先是使用位翻转蝶形运算的kernel函数：

```
__global__ void rev(cuDoubleComplex *a, int *rev, int n) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = id; i < n; i += blockDim.x * gridDim.x) {
        if (rev[i] > i) {
            auto t = a[i];
            a[i] = a[rev[i]];
            a[rev[i]] = t;
        }
    }
}
```

通过并行化的方式，我们加速了位翻转的过程。

通过观察串行过程的主体循环，我们可以发现在不同 `l` 之间的运算是相互依赖的，而在同一个 `l` 之间的运算是互相独立的。因此我们可以并行优化这部分代码。

首先，观察到对于不同的 `j` ， `w` 的值不同。若是每次都计算 `w` 的值，即使是令$w = (cos(j\pi/l), sin(j\pi/l))$，同一个 `w` 值也将被重复计算$\frac{n}{l}$次。总共就重复计算了$n \log n$次。这是一个很大的开销。

因此我们可以预先计算出所有可能的 `w` 的值，存储在一个数组中，然后在kernel函数中直接使用这个数组。观察到 `w` 的值是周期性的，且下标范围为$[0, l-1]$，因此我们可以使用一个大小为 `l` 的数组来存储 `w` 的值。

```
__global__ void init_w(cuDoubleComplex *w, int l, int tp) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = id; i < l; i += blockDim.x * gridDim.x) {
        w[i] = make_cuDoubleComplex(cos(i * M_PI / l), tp * sin(i * M_PI / l));
    }
}
```

预处理完 `w` 的值之后，我们可以直接使用这个数组进行计算。

```
__global__ void fft(cuDoubleComplex *a, int l, int t, int n,
                    cuDoubleComplex *w) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    for (int I = id; I < (n >> 1); I += blockDim.x * gridDim.x) {
        int j = I & (l - 1);
        int i = (I >> t << (t + 1)) | j;
        cuDoubleComplex x = a[i];
        cuDoubleComplex y = cuCmul(a[i + l], w[j]);

        a[i] = cuCadd(x, y);
        a[i + l] = cuCsub(x, y);
    }
}
```

如上所示，由于我们只需要枚举一半的下标，因此我们可以将 `I` 的范围缩小到$[0, n/2)$，并通过位运算来计算 `i` 和 `j` 的值。

主函数中，我们先调用 `rev` 函数，然后循环枚举 `l` 的值，调用 `init_w` 函数初始化 `w` 数组，然后调用 `fft` 函数进行计算。具体代码如下：

```
GPU::rev<<<GRID_DIM, BLOCK_DIM>>>(a, rev_gpu, N);
for (int l = 1, t = 0; l < N; l <<= 1, ++t) {
    GPU::init_w<<<GRID_DIM, BLOCK_DIM>>>(w, l, 1);
    GPU::fft<<<GRID_DIM, BLOCK_DIM>>>(a, l, t, N, w);
}
```

3. 性能对比

我们令 `GRID_DIM=1024` ， `BLOCK_DIM=1024` ， `N=1<<20` ，分别测试了cpu串行版本和gpu并行版本的性能。
程序输出如下：

```
CPU time: 0.50628s
CPU copy to GPU time: 0.06041s
GPU time: 0.005739s
GPU copy to CPU time: 0.002197s
```

可以看到，不计数据传输的时间，计算的加速比大约为88倍。特别的，实际上时间主要的瓶颈在于cpu到gpu的数据传输，约占了总时间的90%。因此，在GPU并行程序的编写中，我们应当尽量减少数据的传输，以提高性能。

三、实验总结

通过本次实验，我们学习了cuda编程的基本知识，并实现了一个并行化的fft算法。通过对比cpu串行版本和gpu并行版本的性能，我们发现gpu并行版本的性能提升了很多。在实际应用中，我们应当尽量减少数据的传输，以提高性能。