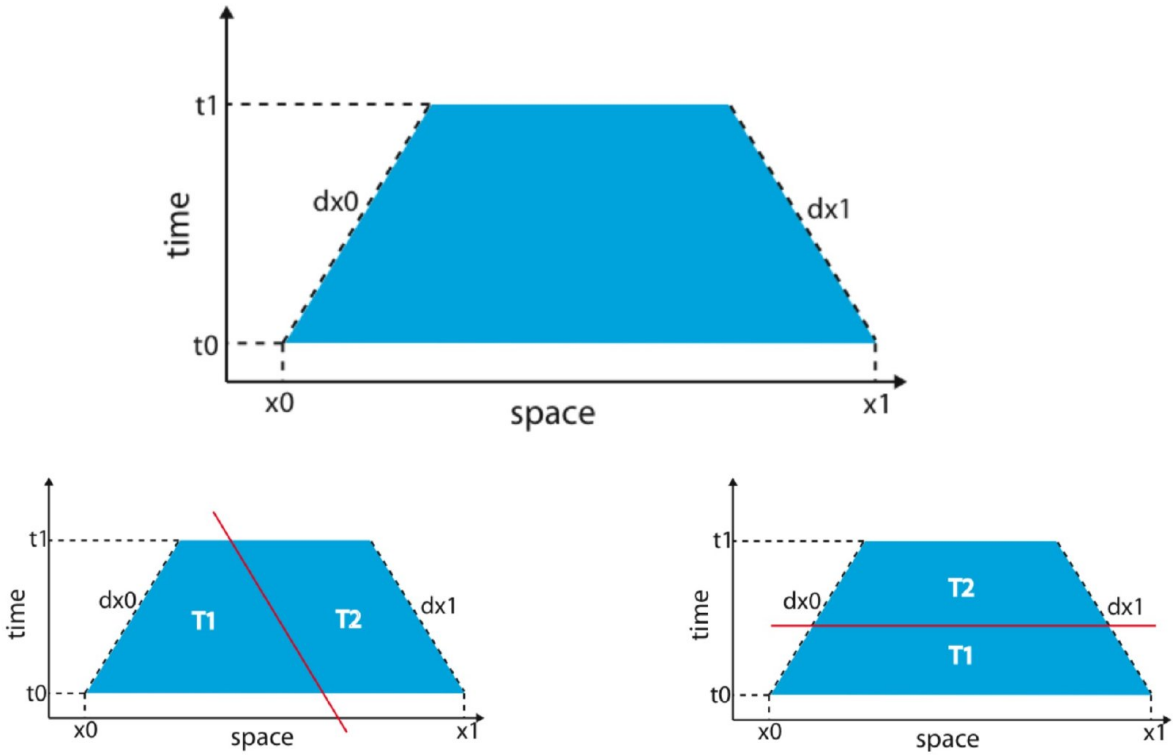
高性能计算实验第二次作业-stencil

> 完成人：马少楠 2018310823

算法介绍

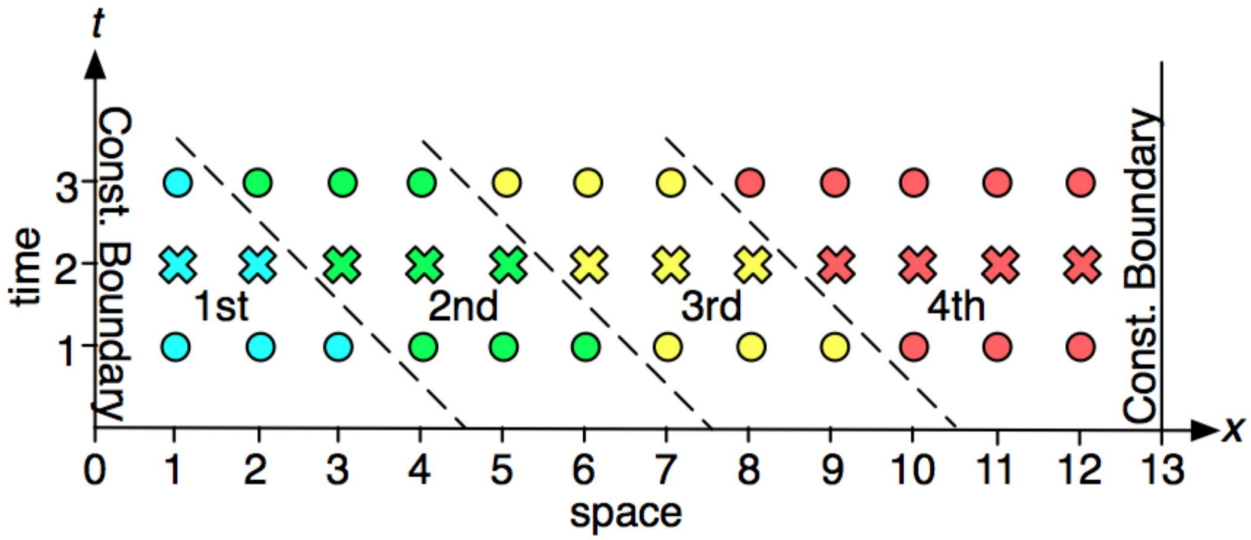
`stencil`是高性能计算中的一个经典问题，有许多种算法用于优化计算这个问题。

oblivious



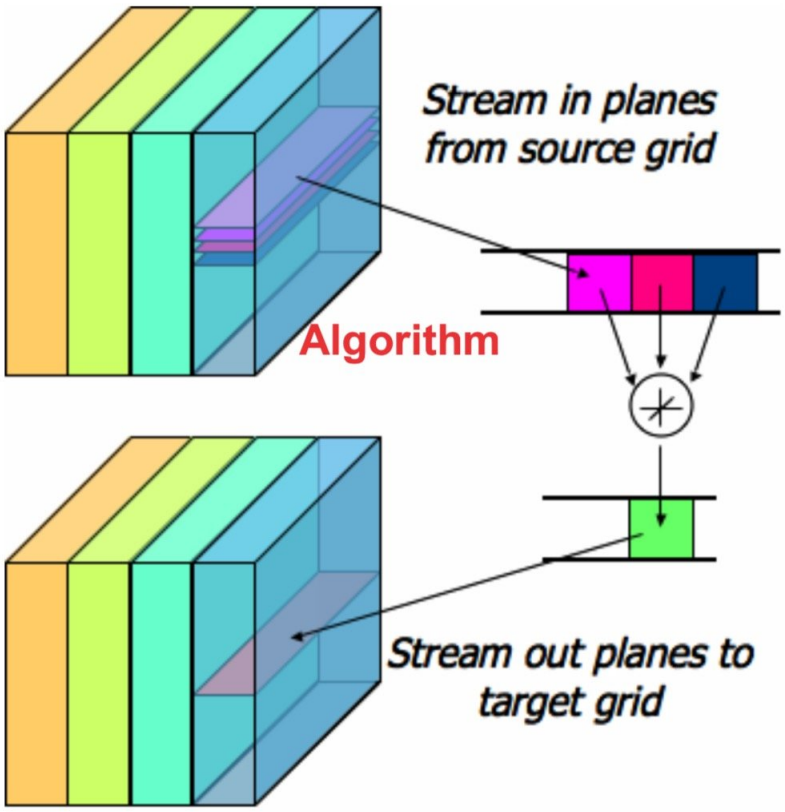
`oblivious`对问题规模进行递归划分，当划分到一个阈值以下时，进行计算。这个方法对缓存无关，所以不需要针对缓存进行调优，需要对阈值进行选取。

timeskew



通过对时间维度斜向划分，可以在`cache`中复用之前计算的数据，有利于性能的提高。对于每一个更靠后的`t`，它需要使用到的前一个时间的数据已经缓存在`cache`中了，会有性能的提高。

circular queue



循环队列算法通过复用三个`read`队列，然后只有首次读和最后时间的写需要学到原先的数据中，中间数据不需要同步，拥有较好的并行性，但是存在较多的冗余计算。

算法实现和性能测试

openmp版本

实现了`naive`，`oblivious`，`timeskew`，`circqueue`四个方法。在计算的部分，使用了`AVX2`指令集，编译优化选项使用了`Ofast`，`xavx2`。

- `naive`仅仅实现了`4`层循环，分别是`t,z,y,x`，然后针对`z`这轮循环使用`openmp`进行多线程并行。
- `oblivious`实现了三维的递归算法，在最终计算的部分使用`openmp`进行多线程并行。
- `timeskew`实现了对于时间进行平行四边形划分的算法，同时进行了分块，在内层计算的位置使用`openmp`。
- `cirqueue`使用了三个`read`循环队列，以及一个`write`队列，通过复用队列，在计算过程中不需要交换数据，在计算`write`队列中的值使用`openmp`。

上述`4`个算法的实现分别保存为`stencil-openmp-naive.c`，`stencil-openmp-oblivious.c`，`stencil-openmp-timeskew.c`，`stencil-openmp-cirqueue.c`，都通过了`test.sh`的正确性测试。

实现细节

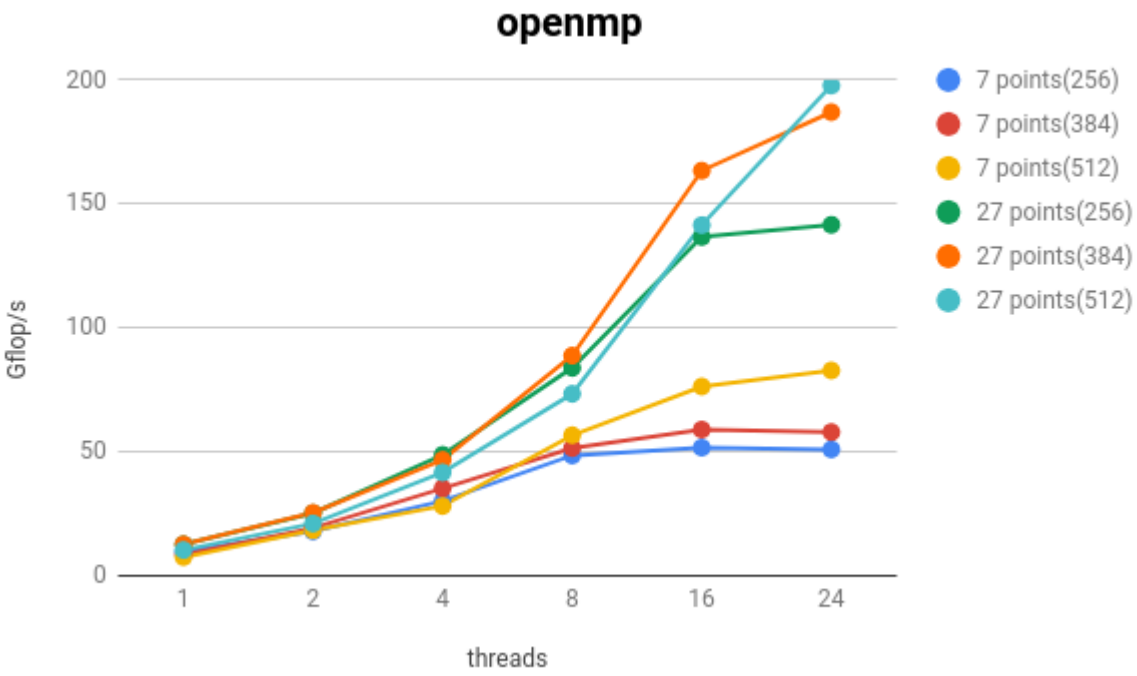
实现的最佳性能是`timeskew`算法的，保存为`stencil-optimize.c`，`cirqueue`优化的性能不高。

实现中，先对`y`方向分块，针对维度为`256`的数据，`y`的块大小为`16`，维度为`384`的数据，`y`的块大小为`12`，维度为`512`的数据，`y`的块大小为`8`。

性能

使用`benchmark.sh`进行`1,2,4,8,16,24`线程来进行计算（仅放`optimize`的性能，其余版本详情见代码）：

thread num	7 points(256)	7 points(384)	7 points(512)	27 points(256)	27 points(384)	27 points(512)
1	9.618929	8.487505	7.504237	12.615269	12.651874	10.377298
2	17.903097	19.243066	18.400227	25.196041	25.382206	21.053222
4	30.16846	35.242507	28.12548	48.74734	46.74399	41.640615
8	48.505234	51.492906	56.681129	83.858471	88.802418	73.390117
16	51.677098	58.948496	76.338248	136.659417	163.403762	141.427374
24	50.953103	57.955556	82.810308	141.544935	187.028643	197.766919



mpi+openmp版本

`mpi` 主要是针对 `blocking` 的算法进行了实现。实现了两个版本，第一个版本是针对维度 `z` 进行切分，第二个版本是针对在使用 `8` 个节点的时候，使用了三维的划分，对 `x,y,z` 三维都划分成了两块。

三维划分可以有效的减少通信量，比如数据维度是 `size`，第一个版本在 `8` 个节点时候的通信次数为 `14` 次，总通信量为 `14*size*size`，三维划分的通信次数为 `24` 次，总通信量为 `24*size/2*size/2 = 6*size`，通信量减少超过一半。

实现细节

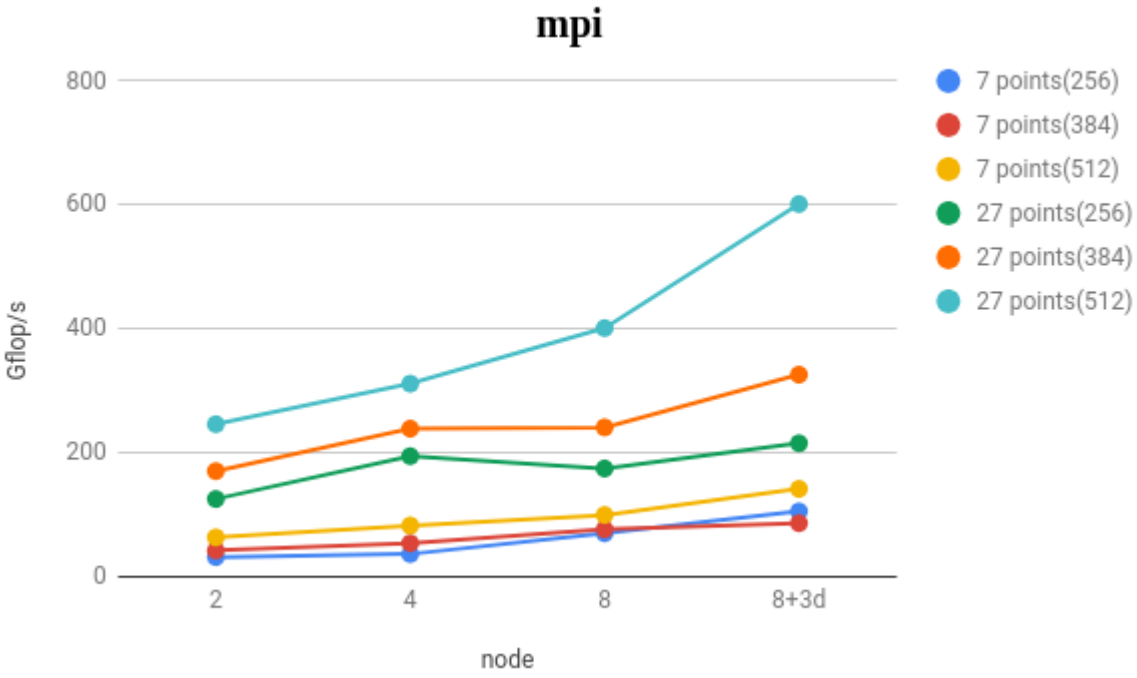
`mpi` 的传输函数使用的是 `MPI_Sendrecv`，因为 `MPI_Sendrecv` 可以有效的解决阻塞模式的消息传递中死锁的问题，写起来较为简单，而且性能非常好，同时测试了非阻塞的 `MPI_Isend` 和 `MPI_Irecv`，然后在等待消息传输结束的时候，先对中心的数据进行计算，实现通信与计算同步，但是得到的效果并不好，所以放弃使用。

对于数据的同步，在 `27` 个点的情况下较为复杂，所以对于每个进程，都确定一个通信的顺序，先和 `x` 方向的进程通信，再和 `y` 方向的通信，最后和 `z` 方向的通信，这样定了一个顺序之后，比如 `27` 点的计算，进程需要的棱上和角上的数据不需要通过单独的传输，只需要通过另一个进程的转发，即可获得，提高了通信的效率，同时降低了编程的难度。

在单个进程中计算，对数据在 `y` 维度上进行了分块，分块的大小和 `openmp` 版本相同。

性能

node	7 points(256)	7 points(384)	7 points(512)	27 points(256)	27 points(384)	27 points(512)
2	31.674837	42.406024	63.429095	125.415374	170.360005	246.111819
4	36.616529	54.062658	82.56446	194.546067	238.90332	311.48178
8	70.078458	76.47223	99.588335	174.574344	240.968135	401.170037
8+3d	105.32339	86.327061	141.801161	215.342549	326.087333	601.378614



cuda版本

`cuda`实现了四个版本，`naive`，`oblivious`，`timeskew`，`cirqueue`。其中`cirqueue`暂时没有写对，因为时间原因没有继续调试。前三个算法实现后发现，性能几乎相同，因为目前实现的是`CPU`单线程的代码，`cache`对于`cuda`程序来说几乎没有影响，所以后续的优化都只在`naive`的实现上进行了，并将`naive`的保存为`stencil-optimized.cu`。

`cuda`计算的程序称为一个`kernel`，一个`kernel`中可能包含一个或者多个`grid`，`grid`中包含了多个`block`，`block`中包含了多个`thread`，`thread`是最终的执行单元，同一个`block`内的线程都会在同一个核心上进行计算，同时一个核心一次会计算一定数量的线程，称为一个`wrap`，默认为`32`。同一个`block`中的所有`thread`共享一块`shared memory`。

`GPU`通过总线和`CPU`相连，首先将数据传输到`GPU`的`global memory`中，但是`GPU`的核心访问`global memory`的性能也不高，所以需要在计算前需要将数据移动到`shared memory`中，对于不同`wrap`移动的数据，移动完成后需要进行线程的同步。

实现细节

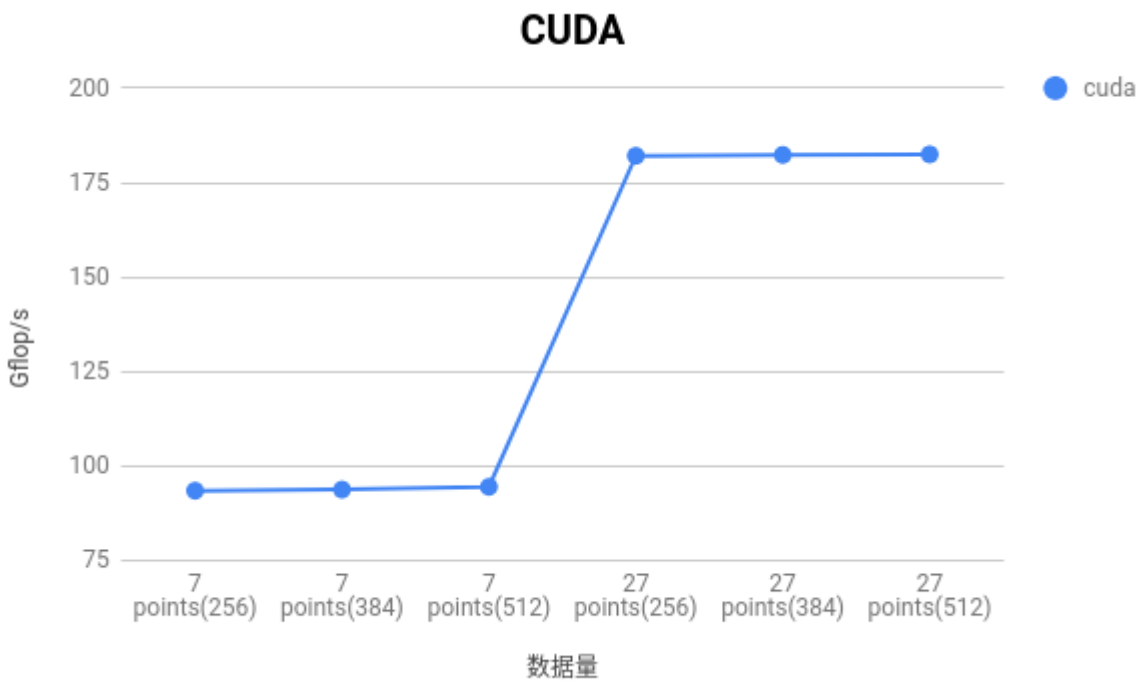
由于`wrap`默认为`32`，所以`block`的维度划分为`32,4,4`，也因为一个`block`中的线程数量不能超过`1024`。

使用了`shared memory`，将一个`block`内线程使用的数据移动进去，性能得到了较大提高。为了减少`index`的重复计算，将`index`中的共同使用的计算提出来，减少计算量，但是这步得到的收益不大。

减少分支预测的逻辑，减少`if`的使用，以及不使用`for`循环（测试得到使用`for`循环会降低大约`30%`的性能）。

性能

7 points(256)	7 points(384)	7 points(512)	27 points(256)	27 points(384)	27 points(512)
93.467185	93.753309	94.491988	182.093654	182.317836	182.473078



参考资料

[1]: openmp文档

[2]: MPI文档

[3]: stencilprobe

[4]: cuda文档