

CS663 - Program Design

This document contains design notes for the various programs created for CS 663/664.

It is not an exhaustive set of documentation for the programs (most of that is in commentary in the source code), but augments those comments.

1 Python Package Structure

The Python code is subdivided into several packages, all named with a short name:

1. AER – Aeronautical Information
2. BDA – BADA equations
3. FUN – Fundamental items, not relying on any of the other Python code
4. GEO – Geographic definitions and transformations
5. PSO – algorithms associated with finding an optimal wind-aided trajectory using Particle Swarm Optimization techniques
6. RTE – Route processing algorithms; at present contains only the parser for the flight aware web site
7. TRJ – routines to build trajectories
8. UTILITIES – main programs for various purposes; other main programs that relate only to a particular package are contained within that package
9. WTH – weather processing, including building the spherical weather products from GRIB files

2 Weather File Processing

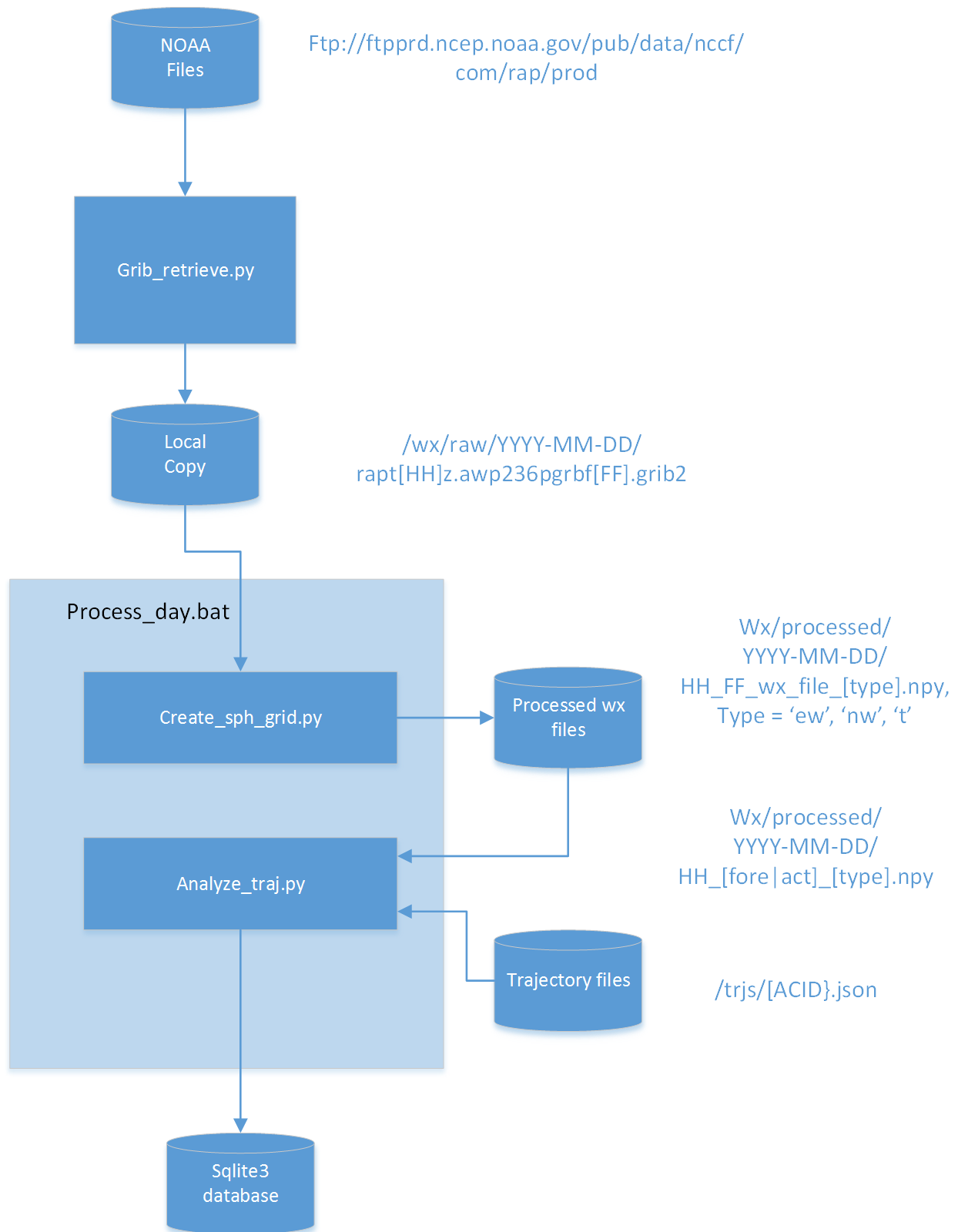
All RAP files downloaded from <ftp://ftpprd.ncep.noaa.gov/pub/data/nccf/com/rap/prod>

These are stored locally for processing in a local directory identified in `wth/config.py`, name `WX_DIR`; typically this is `C:\UMD\CS663\wx`. Under this directory there is a directory named 'raw' (indicating it contains raw data by date), then subdirectories by date; named in the form 2016-11-21.

The files of interest are named `rap.t[HH]z.awp236pgrbf[FF].grib2`, where HH is the hour of the actuals or forecast (00-23) and FF is the relative hour from there; 00 is the actual data from that hour, 01-10 are the forecasts made at that HH for some number of hours in the future.

Weather files are processed separately from the trajectory build program, in order to create a set of weather files per hour. In order to efficiently store the data in numpy arrays, there is one array for each measurement; hence there are arrays for temperature, east winds and north winds. This processed data is stored in `WX_DIR\processed\YYYY-MM-DD`, in files named `HH_FF_[t,ew,nw].npy`. These separate files per forecast hour are combined into a single file per type, name `HH_[t,ew,nw].npy`.

Also in `WX_DIR` is a subdirectory name 'tmp'; this stores temporary files produced during the weather processing.



2.1 Building Processed Files

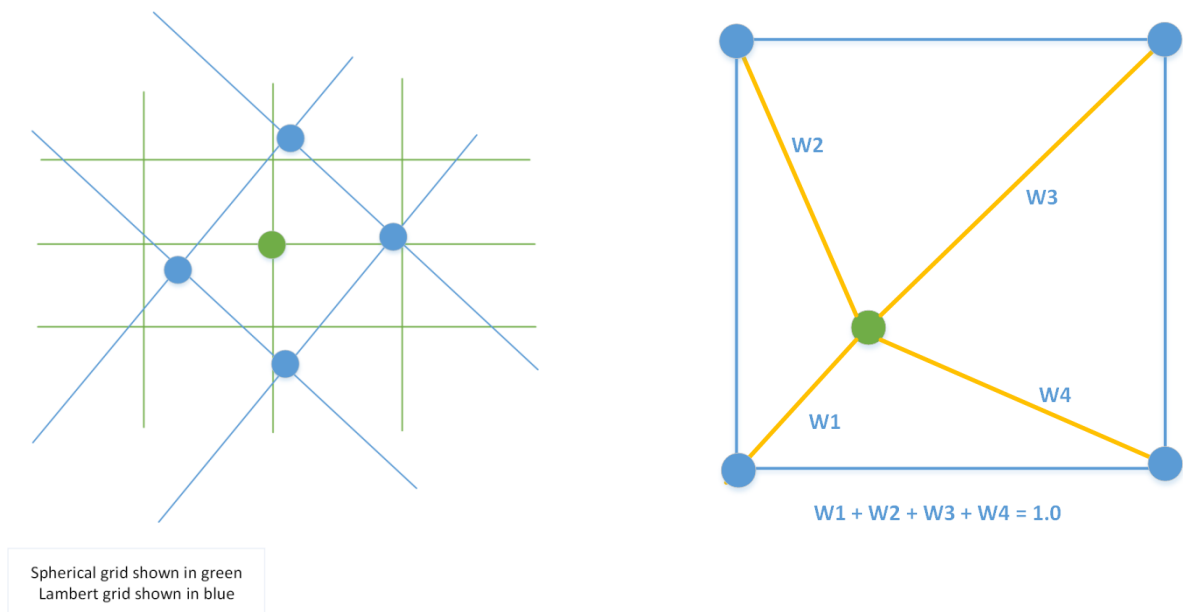
There is a one-time step to create the mapping file from the Lambert grids contained in the grib2 files to the spherical grid used by the trajectory build program. The output from this program is stored in WX_DIR\map; and consists of two files:

- 1) lam_to_sph_map.npy -- this contains entries for each 2D entry in the spherical grid, with a set of information mapping that point to a cell in the lambert grid
- 2) lam_to_sph_info.npy -- this is a python dictionary, listing important information such as grid size and starting point.

2.2 Mapping between Lambert and Spherical

The data from the GRIB2 file is processed by the 'degrib' tool, which converts to geodetic coordinates (lat/long), and lists data by grid point in a lambert grid. Note that in the lambert grid the longitude of all cells in a 'column' is not the same, hence there needs to be a mapping to translate from the lambert grid to the spherical grid.

Zooming in on one particular grid cell, the overlap looks like this (the skew between lambert and geodetic is exaggerated here). The value for the spherical grid point in green is calculated from the four surrounding lambert grid points (in blue). A strict bi-linear interpolation cannot be used because the lambert grid cell is not rectangular. Instead, each of the four lambert grid points are given a weight based on their distance from the spherical grid point; those four lambert points are then averaged based on those weights. Since the juxtaposition of the lambert and spherical grids is fixed, these weights can be pre-computed for each cell and not re-calculated each time a weather file is processed.



2.3 Using Interpolation to find Points in time and space

A weather grid, once read into the python program, is a four dimensional grid, with dimensions of longitude (x), latitude (y), altitude (z) and time (t). The time axis varies from current time (index 0) to some number of forecasted hours (currently 5 are stored). Altitude layers vary by 1000 feet. The size of the x/y grid is set so that a grid cell at the mid-latitude point of the continental US is approximately equal in width (latitude) to the size of the lambert cell; in our case 40 KM.

This gives us a spherical grid of size 327 x 206 x 60 x 6 (assuming one current hour and 5 forecasted hours). There are three values stored, north wind speed, east wind speed and temperature. Hence the total storage consumed by the usual weather grid is 327 x 206 x 60 x 6 x 3 x 8 (8 bytes per value), or 582 megabytes. Thankfully, modern address space sized can accommodate this easily.

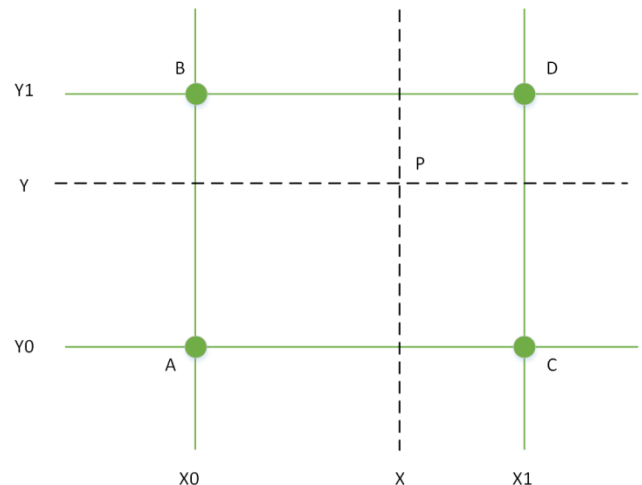
The grid gives values at specific points in space and time. Given an arbitrary point in space and time, the following interpolations are used to find an approximate value at that point:

- 1) First, use a bi-linear interpolation in x and y to find a value for a given latitude and longitude, at the fixed altitude layer below the point in question and at the hour just before the point in question ('hour 0').
- 2) Assuming the altitude given isn't a multiple of 1000 feet, use a second bi-linear interpolation to find a value for the given latitude and longitude at the next higher altitude.
- 3) Use linear interpolation to find the value at the specific altitude needed.
- 4) Finally, assuming the time given isn't on an exact hour, use steps 1-3 to find a second value at the subsequent hour ('hour 1'). Use linear interpolation between the values for hour 0 and hour 1 to get the final result.

The bi-linear interpolation finds a value for a point P with coordinates x and y in the two-dimensional plane. Each value in the surrounding corners (A, B, C and D) are used to find the value for P; a weight is assigned to each corner depending on the distance from P:

$$\begin{aligned}w_a &= (x_1 - x) * (y_1 - y) \\w_b &= (x_1 - x) * (y - y_0) \\w_c &= (x - x_0) * (y_1 - y) \\w_d &= (x - x_0) * (y - y_0)\end{aligned}$$

$$\text{Value of } P = A * w_a + B * w_b + C * w_c + D * w_d$$



3 Flight Intent

Flight intent is obtained from the FlightAware web site, by the fa_parse.py program.

You have to know what flight you are looking for (currently); the parameters to this program are:

- 1) -acid <aircraft id>
- 2) -date <YYYYMMDD>
- 3) -time <HHMM>
- 4) -dep <departure airport ICAO code>
- 5) -dst <destination airport ICAO code>

Airport ICAO codes, for major airports, typically start with "K" and are 4 characters

Flights used for this exercise:

- 1) SWA387 - KBWI => KALB: 20161202, 0320
- 2) SWA3060 - KALB => KMCO (Orlando): 20161202, 1130
- 3) SWA2321 - KMCO => KALB: 20161202, 1400
- 4) SWA994 - KBWI => KSAN: 20161201, 1305
- 5) SWA577 - KSAN => KBWI: 20161201, 2145
- 6) DAL433 – KJFK => KSEA: 20161223, 0025Z
- 7) DAL1997 – KSEA => KJFK: 20161223, 0654Z
- 8) SWA5828 – KSEA => KSAN: 20161223, 0635Z
- 9) SWA5842 – KSAN => KSEA: 20161223, 0520Z

4 Wind Differential Database

The data collected by the 'create_stats' program is stored in a SQL Lite database. This has one record for each Flight/Date/Hour analyzed. Each row has the following columns:

	Column Name	Description
0	Acid	The Flight's call sign
1	Wx_Date	The date of the weather file; e.g 2017-01-03
2	Wx_Hour	The hour of day for the weather file (2,7,12,17 in our case)
3	Avg_Spd	The average speed at hour zero over the cruise segments
4	Absolute Avg Speed	The average of the absolute speed at hour zero over the cruise segments
5	Avg_Act_Fore_Diff	The average difference in speed between hour zero and hour five, averaged over all cusps in the trajectory
6	Max_Act_Cruise_Wind	The maximum value of the head wind at cruise altitude, at hour zero
7	Min_Act_Cruise_Wind	The minimum value of the head wind at cruise altitude, at hour zero

8	Max_Fore_Cruise_Wind	The maximum value of the head wind at cruise altitude, at the proper hour according to the cusp time
9	Min_Fore_Cruise_Wind	The minimum value of the head wind at cruise altitude, at the proper hour according to the cusp time
10	Cruise_Len	The length of the cruise segment, in NMI
11	Cruise_Time	The time spent in the cruise segment, with no wind
12	Avg_Spd_Diff	The average speed difference from using current wind vs. forecasted wind, averaged over all cusps. Negative and positive times at cusps will cancel each other out
13	Avg_Abs_Spd_Diff	The average speed difference from using current wind vs. forecasted wind, averaged over all cusps. Absolute values of the speed at each cusp is used.
14	Cruise_Time_Diff	Time difference of trajectory with current wind vs. using the appropriate forecasted wind, in seconds
15	Fore_Act_Time_Diff	Time Difference in trajectory using forecasted wind 5 hours into the future vs. using actual wind 5 hours into the future

5 Wind Gradients

Both a horizontal wind gradient and vertical wind gradient are calculated for a set of example trajectories. These are created via the `create_gradient.py` program in `utl`.

These are stored in a set of files in the `grads/` directory. Each file is named with the aircraft ID, the date, and the hour of data in that day; for example `grads/DAL433/2017-03-01-02.npy`.

Each file is a 2D numpy array of floats; one row for each cusp in the trajectory (the trajectory created with no wind from the “`trjs`” directory), column zero is the vertical gradient, column one is the horizontal gradient.

Given these definitions:

- 1) LL = lat/long of current cusp
- 2) LL+1 = lat/long of the cusp after the current cusp
- 3) ARW = along route wind
- 4) ARD = ARD of current cusp
- 5) ARD+1 = ARD of cusp after the current cusp

The vertical gradient is defined as:

$(ARW(LL, alt + 1000) - ARW(LL, alt - 1000)) / 2$, in knots/1000 feet. If the altitude at the cusp is less than 1000 feet, no gradient will be calculated.

The horizontal gradient is defined as:

$(ARW(LL+1) - ARW(LL)) / (ARD+1 - ARD)$, in knots/nm. There will be no value for the last cusp.

All wind values are taken from the wind grid, interpolated in the horizontal and vertical dimensions. In order to tell if a cusp is in the climb, cruise or descent segment, the trajectory must be queried.

The “create_gradient.py” program will create the individual gradient arrays for each flight of interest, for each date and hour of interest. These files can then be analyzed.

The “gradient_summary.py” program will do some analysis on the data; finding (for each flight) the one cusp that had the highest vertical and horizontal gradient, and the weather set that yielded the largest average gradient (averaged over all cusps). For the purpose of this analysis, only vertical gradients for climb and descent segments are included, and only horizontal gradients for level (cruise) segments are included.

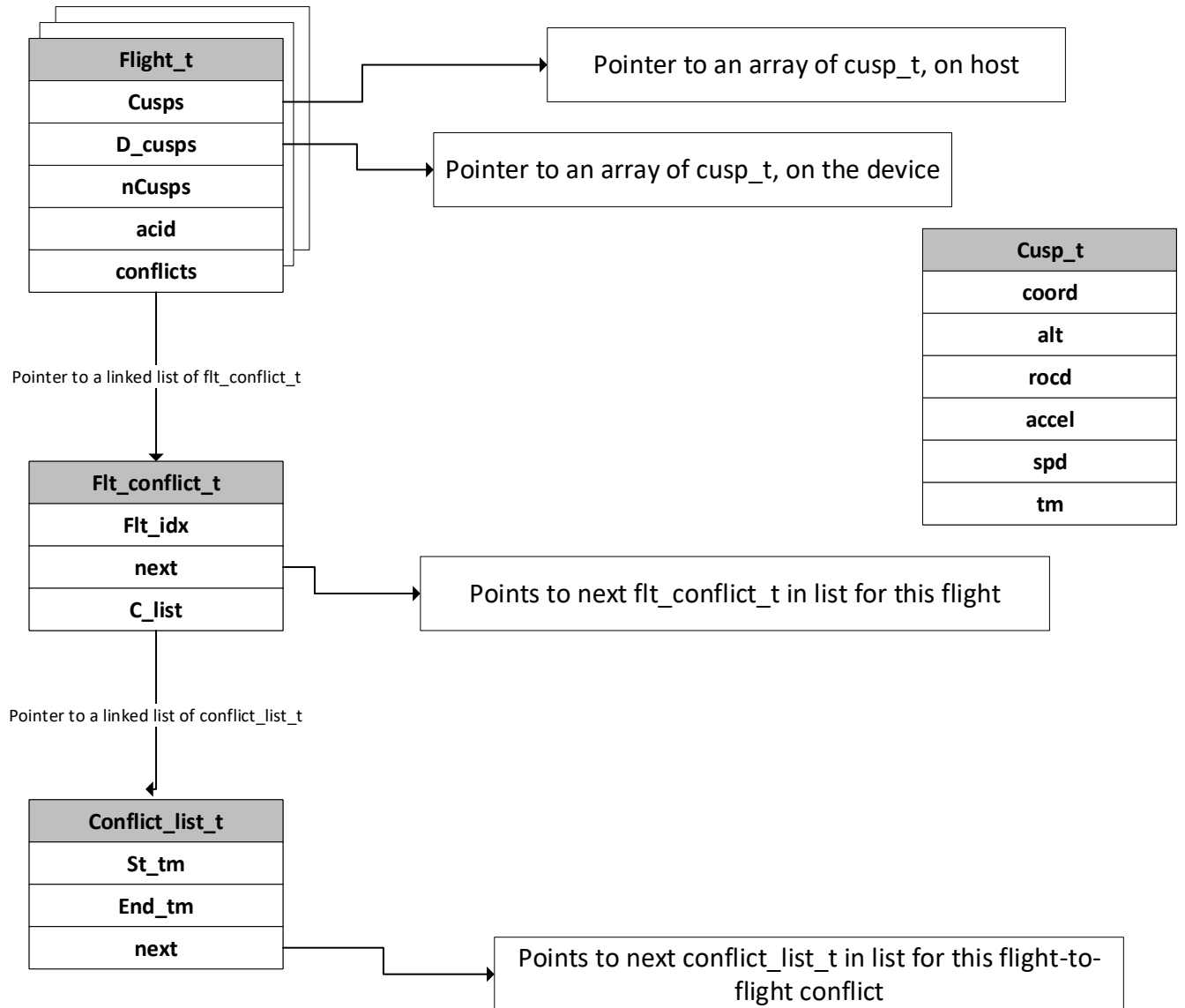
Flights used for this exercise:

Num	ACID	Dep	Dest	Date	HHMM
1	AAL1147	KMIA	KLAX	3/23	0149
2	AAL134	KMIA	KSEA	3/23	0057
3	AAL2046	KMSP	KCLT	3/23	????
4	AAL2245	KMIA	KSAN	3/22	0001
5	AAL23	KJFK	KLAX	3/23	0215
6	DAL1579	KMSP	KBWI	3/23	1350
7	DAL1587	KBIS	KMSP	3/23	1140
8	DAL1938	KMSP	KSFO	3/23	1705
9	DAL2865	KMSP	KSEA	3/23	1355
10	JBU208	KJFK	KPWM	3/23	1403
11	LOF4628	KBIS	KDEN	3/23	1120
12	SWA1493	KMDW	KLAX	3/23	1030
13	SWA1652	KDEN	KPHX	3/23	1255
14	SWA1688	KMDW	KHOU	3/23	1145
15	SWA1715	KMCI	KHOU	3/23	1040
16	SWA1786	KATL	KDEN	3/23	0955
17	SWA2321	KMCO	KALB	12/02	1400
18	SWA314	KDAL	KSEA	3/23	????
19	SWA325	KBOI	KOAK	3/23	1245
20	SWA336	KBOS	KHOU	3/22	1550
21	SWA4650	KBOI	KPHX	3/22	1755
22	SWA561	KMDW	KSAN	3/22	0020
23	SWA5828	KSEA	KSAN	12/23	0635
24	SWA653	KMCI	KOAK	3/22	1635
25	SWA806	KMDW	KFLL	3/23	1105
26	SWA994	KBWI	KSAN	12/01	1305

6 Conflict Probe

Conflicts found (regardless of the version) are stored in the following data structure:

Flights: array of `flight_t`,
statically defined in “main”

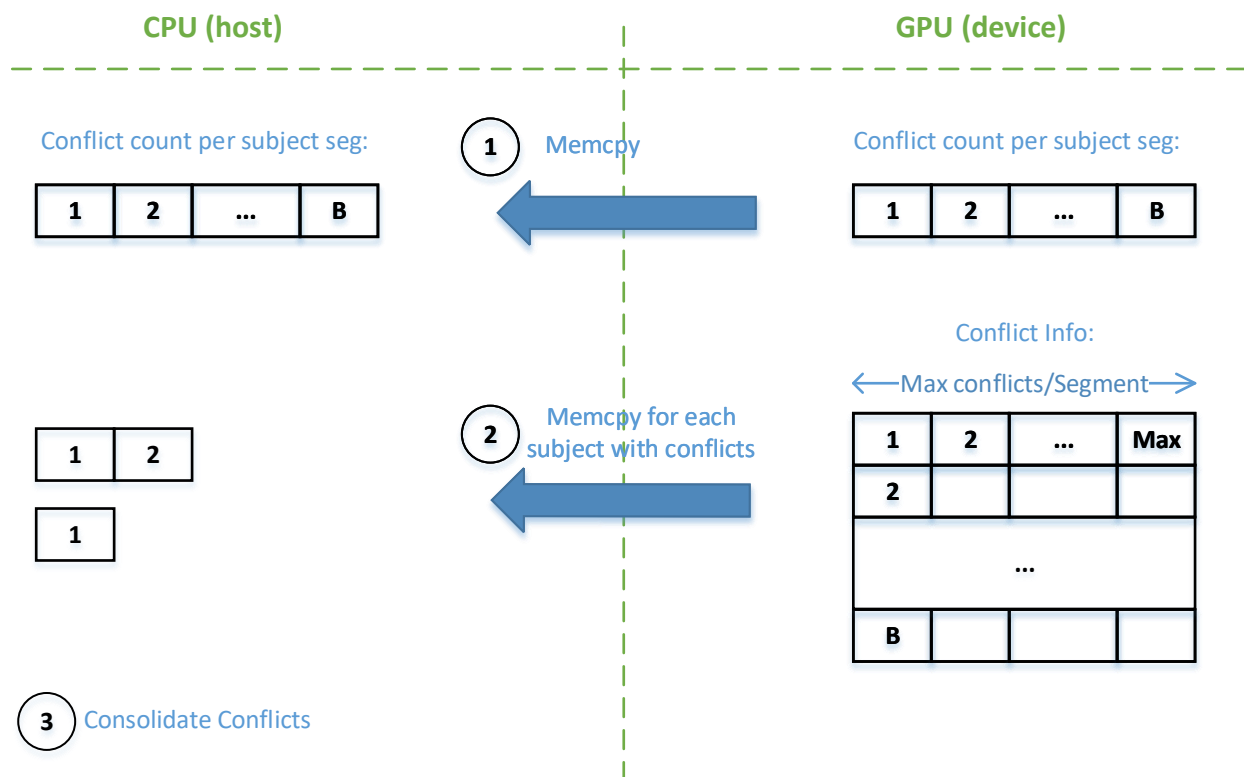


Version 2 of conflict probe performs all the segment comparisons of one flight with another flight in parallel.

The number of grids used is the same as the number of segments in the “new” (subject) trajectory. The number of threads is the number segments in the object trajectory. Each kernel compares one subject segment to one object segment.

There is one conflict array in thread-shared storage to accumulate the conflicts for the single subject segment. A thread sync operation is performed by the kernel, which will sync an operation of one subject segment to all object segments. At the conclusion of the sync; thread ID 0 will accumulate the conflicts found into global storage, and place the number of conflicts for that subject segment in a global array (that global array has an entry for each subject segment).

Once all kernels have completed, the conflict count array is copied back to the host. Then for any non-zero count, that many conflicts are retrieved from global storage. Hence only real conflicts are copied back to the host. This is depicted below:



Version 3 changes this so that there is just one memcopy at the conclusion of the process, not one per subject segment in conflict. This is done at the expense of adding another kernel call to accumulate all conflicts in general purpose device storage. This looks like:

CPU (host)

GPU (device)

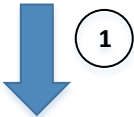
Conflict count per subject seg:

1	2	...	B
---	---	-----	---

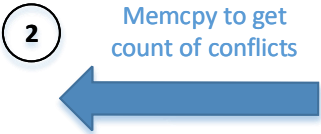
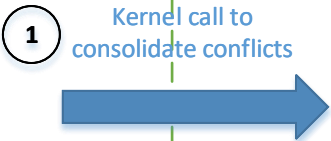
Conflict Info:

←Max conflicts/Segment→

1	2	...	Max
2			
...			
B			



1	2	...		Max
---	---	-----	--	-----



1	2	...		Max
---	---	-----	--	-----