PROGRAMMING TIPS

1) for absolute beginners: start programming with Octave M-scripts (high-level, ultra-easy-to-learn, scientific language),

2) useful links for smartphone/PineA64+/OrangePi C++ users:

-original C++ official site, and online compiler:

http://www.cplusplus.com/
http://en.cppreference.com/
http://www.cpp.sh/

-tutorials with online compiler on multicore platform session:

https://www.tutorialspoint.com/cprogramming/
https://www.tutorialspoint.com/cplusplus/index.htm
http://www.tutorialspoint.com/compile_cpp11_online.php

-great books etal. prof. Stroustrup for beginners and advanced programmers:

"Programming principles and practice using C++", "C++ programming language"

3) for data processing beginners (GB's of scalars and vectors data processing experience) in sequential C++, parallel CUDA C:

-use **array** or **vector** container (read&process chunks of data via cache, with correct CPU memory bandwidth usage); There is a performance gap between static vector as smallest container and any other container in C++. For psuedorandom bijection accesses use **map** container (unordered_map is faster, map – self sorting during creation via iterator); quite intuitive and good results are for vector< vector< float > > - vecD (vector in D-dimensions, first dimension is allocated in continous virtual memory address – physicall memory continuity is platform dependend),

-typical today's 2 or 4 channel RAM architecture provides best efficiency for 4B datatypes,

-acceptable results will be for bidirectional list<vector<float>> if different size vectors are often added/deleted,

-**UNCOMPRESS** to RAM:

-tip from prof. Starzynski from Warsaw University: „faster for bigger data"

-applicable from few MB's data size, can reduce used PCIe bandwidth in CPU-GPU transfers, and any network computations transfer,

-**SORT** = organise data:

-easier and faster to processing

-quite fast to obtain statistics (min, max, median, etc.)

-**NORMALIZE** data:

-one way to normalize is get median value from mean of sorted data and store it separately and as second step subtract it from data. If it does fit in float storage acceptable range get min element as 0 and divide each element by max element normalize to 1, (note that norm[-1, 1] step is lossy if max element is much greater than median - use mean instead). Consider making variables as struct similar to (during parallel calculations it should make better job than double calculations – just check variable range in code execution, it is typical effort in Digital Signal Processor fixed point computations):

```
struct floatMedian
{
    float professionalComputations [ const unsigned bigEnoughForParallel];
    long medianExtracted += calculatedMedian; //initalize with 0
    uint8_t medianExtractedOverflow;
};
```

-note median drawbacks – there are some other tricks like trimmed mean

-one can use fastest datatypes which are commonly FLOAT (uint16_t are basicly for counters – use auto, or iterators), and store only an additional median value to on-air normalized data

-FLOAT considerations:

-data storage precision: +-1000 variable value (better are +-100)

-accumulative computation accuracy max iteration = 100

-variable comparisions (err: 1.1==1.1!!!):

global float floatErr = 10e-5;

(varVal – floatErr < 1.1)&&(varVal + floatErr > 1.1)

pro: use CPU register described in 06_registerGlobal

-note that float pointer size is doubled than float datatype size on 64bit architecture:

sizeof(float*) = 8

sizeof(float) = 4

sizeof(uint16_t) = 2

-instead of float, the size of variable double representation size is doubled, but computation capabilities of double variables are not halfed (<4 times),

-keep vectors normalized and store its median value as separate variable

-greatest range for floats is [-1.0f, +1.0f] (23bit - minimal storage error)

-periodically normalized vector of floats is more accurate than any other floating point variable (including long double)

-**FRAGMENT** data:

-only not too small fragments of data with low no. of programming instructions are fast

-commonly only small fragments are trivially attainable

-fragmented data are simpler to compute in parallel:

PARALLELIZE computations:

-quite simple sequential parallelism on multicore CPU: GNU-please-cite-Parallel scripts,

-quite simple parallelism on GPU: accelerated libraries like Thrust (thrust::device_vector - omit expensive transfers for small data size),

-computations of vectors with size bigger than 8000 elements are faster only on GPU (for AMD CPU fans – please check single thread CPU and GPU efficiency for big enough data – you will be surprised with results – the GPU/CPU single thread efficiency gap is small, and depends on processing type),

-keep computations only on GPU, use CPU for communication,

-**Time To Market considerations** – cost of computing Octave easto-to-write script few times during night or writing an OpenCL optimized application for week and obtain results in seconds (quite rare situation of often used programs). Please note, that only open, free/cheap, well written, fast/optimized, low-complexity programs will stay on market for long time.

-**Look up Tables –** usage of static vector containing prec-computed values highly reduces overall amount of computations. Check your program for multiple calculations of **2+2=4**! (save correct partial results)

4) please choose low complexity algorithm, corectly use your hardware capabilities, bench/profile your code to get tuning benefits. Keep last two working versions of program.

5) C++ offline documentation could be accessible via cppman package:

#sudo aptitude install cppman

#cppman -c #offline download / update

#cppman -m true #set cppman instead of #man in #vim

a) #cppman vector

b) #cppman std::vector

c) #man std::vector

6) shell alias is convenient way to make an autofilling in terminal, simply edit bash configuration file (for example „ls -als” shortcut with „ll”):

#sudo vim ~/.bashrc
#alias ll="ls -als"

7) trivial and cheap way to speed up exhaustive I/O traditional programs (CPU computations on large set of small files) is to put your OS in Virtualbox. Place such VM on RAMDISC, described in LINUX_tips repo. Bigger problems should be solved with usage of GPU and RAID00 SSD's (mdadm program RADI0 on hardware RAID0 disc controllers),

8) some big amount of universal memory could be cheaply achieved with used SATA disc and USB – SATA adaptor,

9) please make usage of your compiler flags for example makes almost linear compilation time speed up:

#gcc -mtune='native' -march='native' -Ofast -pipe
#make -j`nproc`

10) heavy – load traditional computations could be efficiently provided with usage of past generation servers (loud equivalent to hairdryer) and remote acces with network – there are some tricks like for example non-filtered low voltage electrical grid GE adaptors for home usage,

11) do not trust too much to your hardware capability stereotypes. Annually one should provide test computer components (RAM partial disabilities, HDD bad sectors, registers quantum tunneling errors at overclocked CPU's over ~4GHz, et cetera). Each new device configuration start tests should be focused on truly each memory levels throughput (caches, RAM, GPU global memory, PCIe slots, network bottlenecks, et cetera) and GFLOPS capabilites (or GIOPs for GPU fixed point arithmetics – AMD GPU's generally performs better). Author personally uses these basic and universal benchmarking tools:

-x86 memtest liveUSB for CPU caches and overal RAM capabilities (it provides detailed benchmark of different block size read and write accesses),
-CUDA-Z for GPU memory and computing capabilities, PCIe bus throughput,
-iperf for network throughput,
-Ubuntu liveUSB for disks and disk arrays throuput via Disks->Benchmark Partition,
-linpack GFLOPS for single core CPU computing efficiency (please refer to 78_linpackGFLOPS),