Autovectorisation in CMS: lessons learned, tools and techniques

T. Hauth, V. Innocente, D. Piparo

Annual Concurrency Forum Meeting

Contents

- Autovectorisation
- Vertex reconstruction in CMS
- Lessons learned and conclusions

Autovectorisation

- "Vectorisation": usage of CPU's vector registers to speed up computations (simplest form of data parallelism)
- Several techniques and tools available: Cilk++, superword level parallelism (SLP), intrinsics, autovectorisation …
- "Autovectorisation": automatic procedure put in place by the compiler to transform regular C++ code of loops into machine code invoking vector instructions
  - Rather mature in GCC 4.7 and other compilers
  - Maximum portability (MIC, ARM): the compiler does all the work

How to efficiently vectorised loops

It's necessary to "help" the compiler. Some general guidelines:

- Use countable loops (number of iterations known at runtime before start)
- Avoid non-contiguous memory access
- Single entry and exit
- Simple data dependencies
  - E.g. avoid to read variable and write it in a subsequent iteration
- Do not call functions (unless actually inlined)
- Limit usage of branches (some may be tolerated: mask assignment)
- Prefer Structures Of Arrays (SOA) to Arrays Of Structures (AOS)

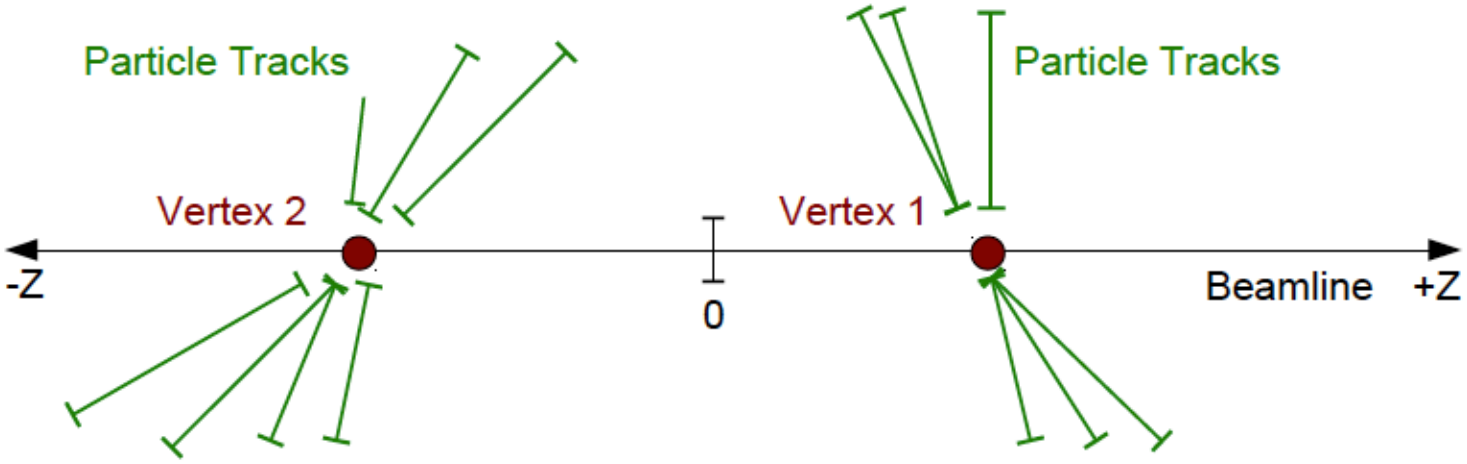
… Keep it simple! Think in C, everything is an address in memory

To obtain maximum performance:

- Use single precision instead of double
  - nominally 2x faster, true for vectorisation in general)

Test-case: vertex reconstruction in CMS

- CMS reconstructs primary vertices' z coordinate with deterministic annealing (DA)
  - It took 3-5% of reconstruction time (a lot of money)
  - Existing implementation had several loops and C arrays filled with information from the existing data structures
  - Perfect case for autovectorisation!



Test-case: vertex reconstruction in CMS

- DA uses exp function
  - First, calls factorised out of the loops
  - Then, an inline, approximate and autovectorisable exp function implemented
- Results identical to scalar version:
  - This particular algorithm is forgiving, not guaranteed in general

In production since more than a year

Version	DA Runtime [s]	Ratio
Regular	29.64	1.0
Autovectorised	19.96	0.74
Autov.+ VDT exp	11.46	0.43

Factors *can* be gained

T Hauth *et al .: Conf. Ser.* 396 0520652012 *J. Phys*

Lessons learned

- Autovectorisation is a powerful strategy to exploit vector units
  - Rely purely on the compiler, highly readable code
  - Complementary to other techniques
- Fragile: an unintended modification (an if, a call) can break it
- Effort required: mix of knowledge of the physics involved, codebase and programming skills
  - For DA simple data structures already in place made life easier
  - True in general for the non-trivial cases (basically absent in scientific code)
- Reducing precision (double to float) is advisable but not trivial
  - Validation effort (and tools) needed not to jeopardise physics performance
  - But: a factor 2 in speed is at stake
- Vectorisation pre-condition: simple data structures
  - Re-write of the present widespread complex OO ones, prefer SOAs

Conclusions

- Data structures do matter
  - True in general for vectorisation
  - Most prominent show-stopper
- Autovectorisation guarantees speed and maximum portability relying on the compiler only
- Its fragility suggests, if the problem allows it, to turn to specific libraries offering common building blocks (e.g. algebra, fitting, math)
  - This does not exclude to use autovectorisation *within* libraries