Intermediate deadline feedback

LIÈGE université

INFO0939: High Performance Scientific Computing

Fall 2025

Clean and safe code ≠ efficient code

Be careful not to sacrifice performance by adding unnecessary safety checks. Even simple if statements inside performance critical loops can prevent the compiler from performing vectorization

If you rely on `assert()` for safety during development, remember that it generates runtime checks. These checks often block vectorization and other loop optimizations. To avoid this in Release builds, make sure to compile with `-DNDEBUG`

You can use compiler flag to get vectorization information:

- `-fopt-info-vec`: basic vectorization info (vectorized loops)
- `-fopt-info-vec-missed`: loops that failed to vectorize
- `-fopt-info-vec-all`: everything: success, failure and “reasoning”

OpenMP: are you parallelizing the right way?

Parallelizing the time loop does not make sense because each time step depends directly and immediately on the fields from the previous time step. The update equations are strictly sequential in time: you cannot compute fields at $t + 1$ until the fields at t are fully known

If your OpenMP implementation requires use of constructs like `barrier`, `master`, or `single` just to produce correct results, it's a strong sign that the design choices might not be the right one

---

When adding optional clauses to OpenMP `parallel` or `for` constructs, ensure that each one is truly justified. It's important to ask your self (and measure) whether they actually improve performance

CUDA: do not introduce unneeded synchronizations

Although GPU kernel launches are asynchronous with respect to the host, their execution order is guaranteed within a given CUDA stream (the default stream in your case). This means that while the CPU continues running without waiting for each kernel to finish, the kernels themselves will still execute sequentially in the order they were issued, as long as they are placed in the same stream

Because kernel execution order is guaranteed within a single CUDA stream, inserting explicit synchronizations between every kernel launch (`cudaDeviceSynchronize()`) is both unnecessary for correctness and detrimental to performance:

- It forces the CPU to pause until all previously launched GPU work is complete which introduces a significant synchronization overhead
- Each kernel launch has an inherent latency. This latency can be hidden by overlapping kernel launches with other work on the GPU

CUDA: block size and memory accesses

As a remainder, consider the following information:

- Memory accesses are most efficient when consecutive threads in a warp (32 threads) access consecutive memory locations (coalesced memory access)
- A cache line is 128 bytes (four 32 bytes sectors)
- When using 2D blocks, a warp is formed along `threadIdx.x` first (fastest-varying index) then along `threadIdx.y`

When choosing your block size, ask yourself:

Does my block size allow each warp to access memory contiguously and fully utilize cache lines?