Forward modelling of acoustic waves on HPC architectures

Andreas Hadjigeorgiou and Eric Verschuur
ENGAGE workshop on HPC and Data Science

The Cyprus Institute, 22 June 2023

Content

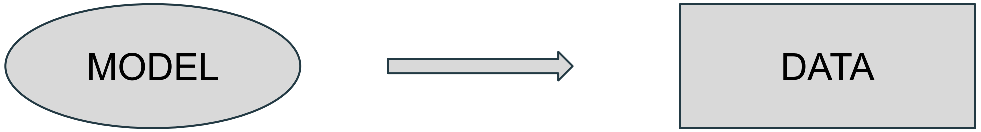
Forward modelling

Forward modeling refers to the process of **simulating or predicting the response of a physical system** or phenomenon based on a known or hypothesized model. It involves using **mathematical equations and computational techniques** to simulate the behavior or **outcome of the system under specific conditions**. In various scientific and engineering fields, forward modeling is employed to understand and predict the behavior of complex systems, validate theoretical models, **interpret experimental data**, and make predictions about real-world phenomena.

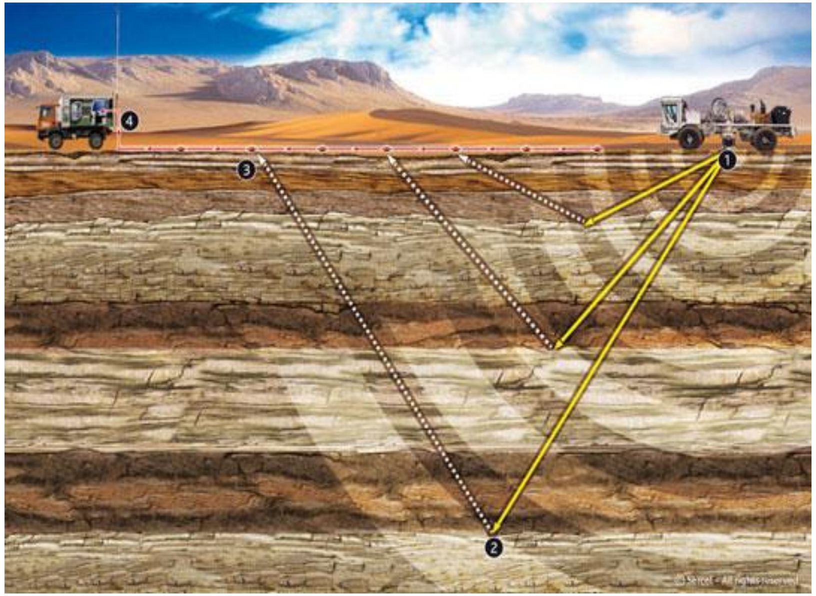
We know the model

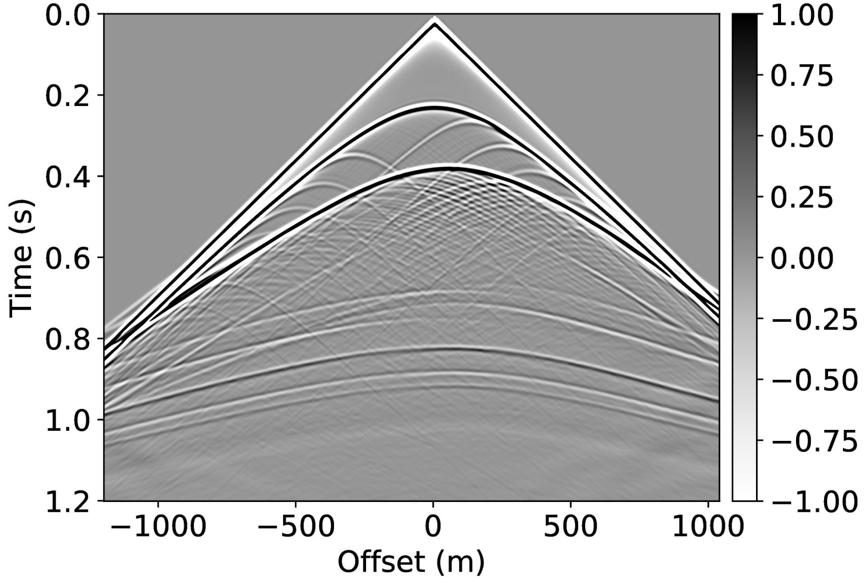
We know the initial conditions

We know the equations that govern the system

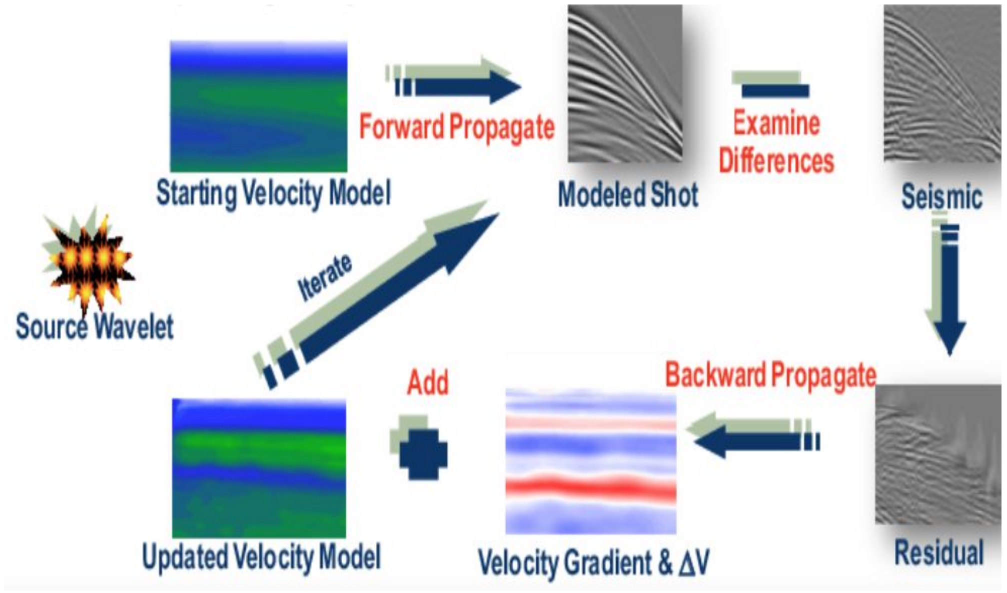


If we can simulate data that fit our observations (seismic data), then we have a good estimate of the underlying model.





Seismic imaging relies on efficient and accurate modelling of acoustic waves



Kapoor, Sanjeev et al. "Full Waveform Inversion Around the World." (2013).

Forward modelling and imaging rely on the same principle: wavefield propagation based on acoustic wave equation.

95+ % of total computation accounts for simulating wave propagation.

Acoustic wave-equation

$$\frac{\partial^2 P(t,x,z)}{\partial t^2} = v(x,z)^2 \left(\frac{\partial^2 P(t,x,z)}{\partial x^2} + \frac{\partial^2 P(t,x,z)}{\partial z^2} \right) + S(t)$$

P(t, x, z): acoustic pressure amplitude at given time and space point

u(x, z): space-dependent velocity of acoustic pressure

S(t): time-dependent source

Finite-difference modelling

2nd order in time: $f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$

4th order in space: $f''(x) = \frac{-\frac{1}{12}f(x-2h) + \frac{4}{3}f(x-h) - \frac{5}{2}f(x) + \frac{4}{3}f(x+h) - \frac{1}{12}f(x+2h)}{h^2}$

Algorithm steps

Attention to the boundaries!
Two-points layer of zeros.

```
Loop over time-steps t:
    Add_source:  P(srcz, srcx) = P(srcz, srcx) + S(t)
    Loop over space-steps z, x:
        Compute Pzz:  Pzz(z,x) =  -5/2*P(z,x) + 4/3*P(z±1,x) - 1/12*P(z±2,x)
        Compute Pxx:  Pxx(z,x) = -5/2*P(z,x) + 4/3*P(z,x±1) - 1/12*P(z,x±2)
        Compute Ptt:  Pnew(z,x) = 2*P(z,x) - Pold(z,x) + ( (u(z,x) * dt / dh)^2 * ( Pzz(z,x) + Pxx(z,x) ) )
    Pold = P
    P = Pnew
```

Placing the algorithm in the context of HPC

Our goal is to develop the algorithm such that we utilize the available resources effectively in order to maximize the performance of computations.

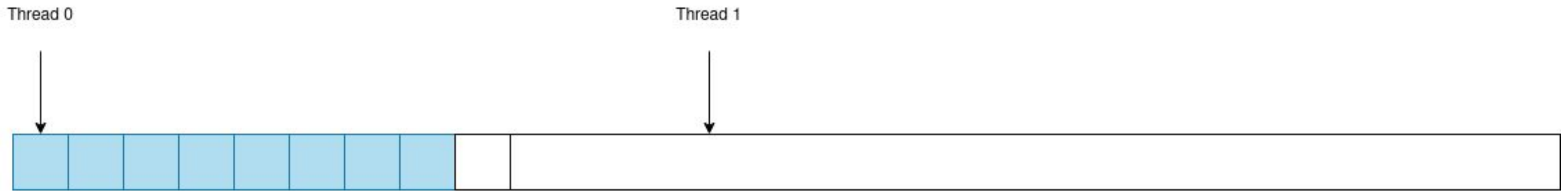
To achieve this we need to “map” the computations properly to the hardware:

1. proper layout of data in memory
2. proper placement of threads

We target CPU and GPU architectures, which have some significant differences!

On CPUs we aim for threads-caching

Thread caching is a concept related to the caching behavior of individual threads in a parallel computing system. In many parallel computing architectures, each processing unit or thread has its own cache, which stores frequently accessed data. Thread caching allows threads to store and retrieve data from their local caches, reducing the need to access slower main memory.

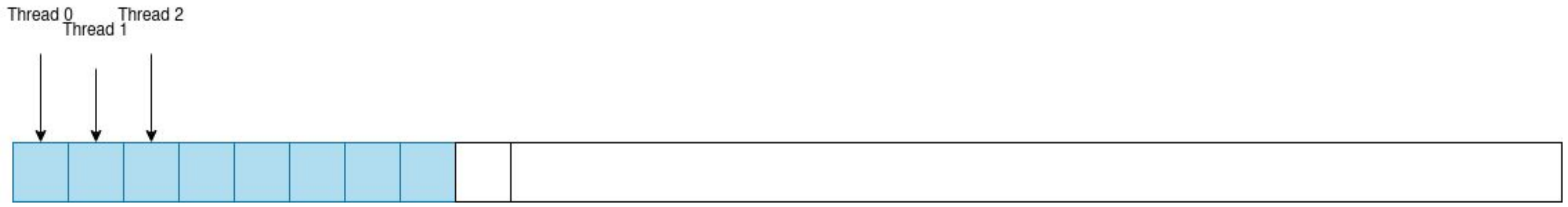


When a CPU thread/core updates a particular memory address, the whole cache-line (64 bits typically) is cached $\Longrightarrow$ the other threads are "informed" (cache coherency)!

This mechanism does not come at zero cost

On GPUs we aim for threads-coalescing

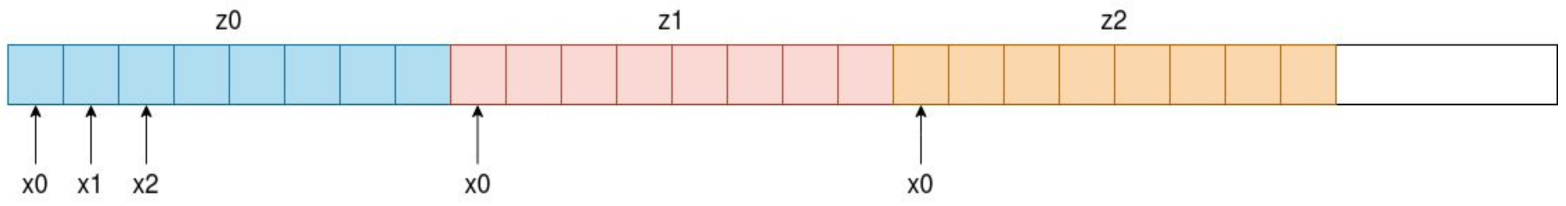
Thread coalescing refers to a memory optimization technique used in GPU programming, specifically in the context of memory accesses by parallel threads in a GPU thread block. In GPU architectures, memory accesses are more efficient when neighboring threads access adjacent memory locations.



On GPU threads are scheduled in warps (32 typically), which operate concurrently only as long as they access consecutive memory places.

Otherwise, computations become sequential

Data layout



We develop a class called `ScalarField` that represents the *wavefields* and the *velocity-model* in our simulation. The class stores the data in a suitable layout in memory.

The layout is common both for the CPU and the GPU!

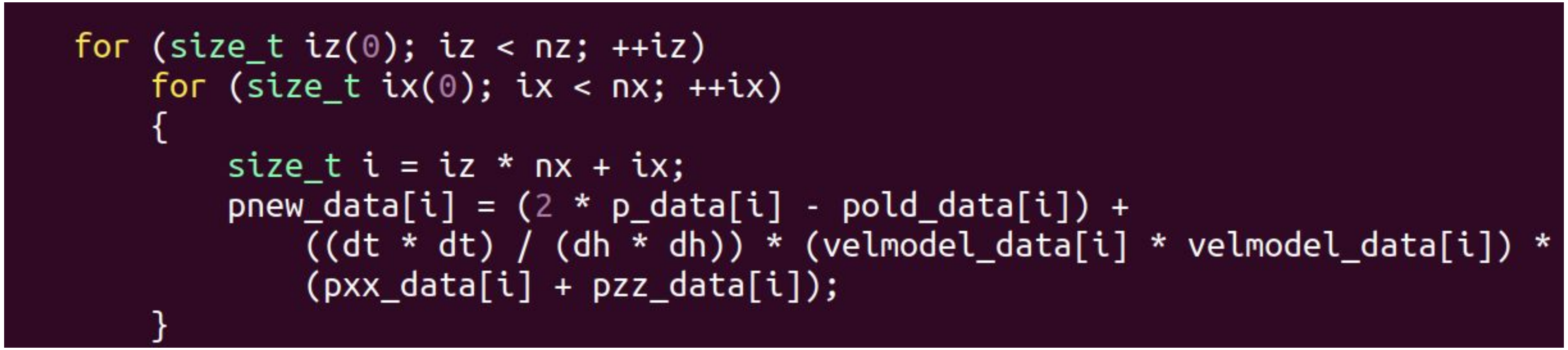
The layout is linearized as follows:
P(iz,ix) -> P[iz*nx + ix]

Finding concurrency

`add_source` is a routine with no parallelism to exploit!

Let’s see the implementation of the time_extrapolation routine, which involves the computation of ***Nz* * *Nx*** points!

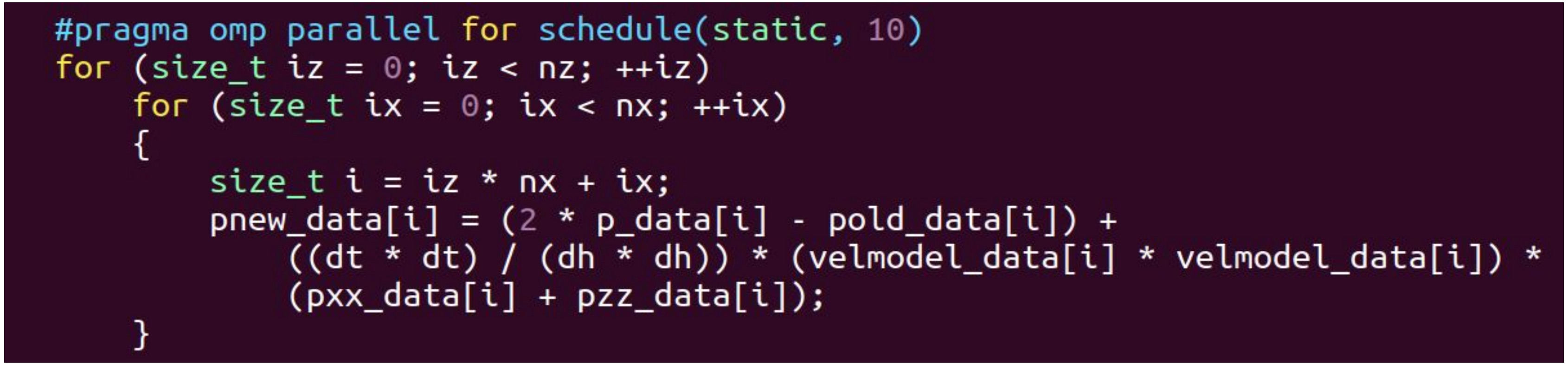
C implementation of time-extrap



```
for (size_t iz(0); iz < nz; ++iz)
    for (size_t ix(0); ix < nx; ++ix)
    {
        size_t i = iz * nx + ix;
        pnew_data[i] = (2 * p_data[i] - pold_data[i]) +
            ((dt * dt) / (dh * dh)) * (velmodel_data[i] * velmodel_data[i]) *
            (pxx_data[i] + pzz_data[i]);
    }
```

- Loop over x is innermost in order to achieve optimal sequential access pattern.
- Simple element-wise operations.
- All computations are decoupled -> trivial parallelism.

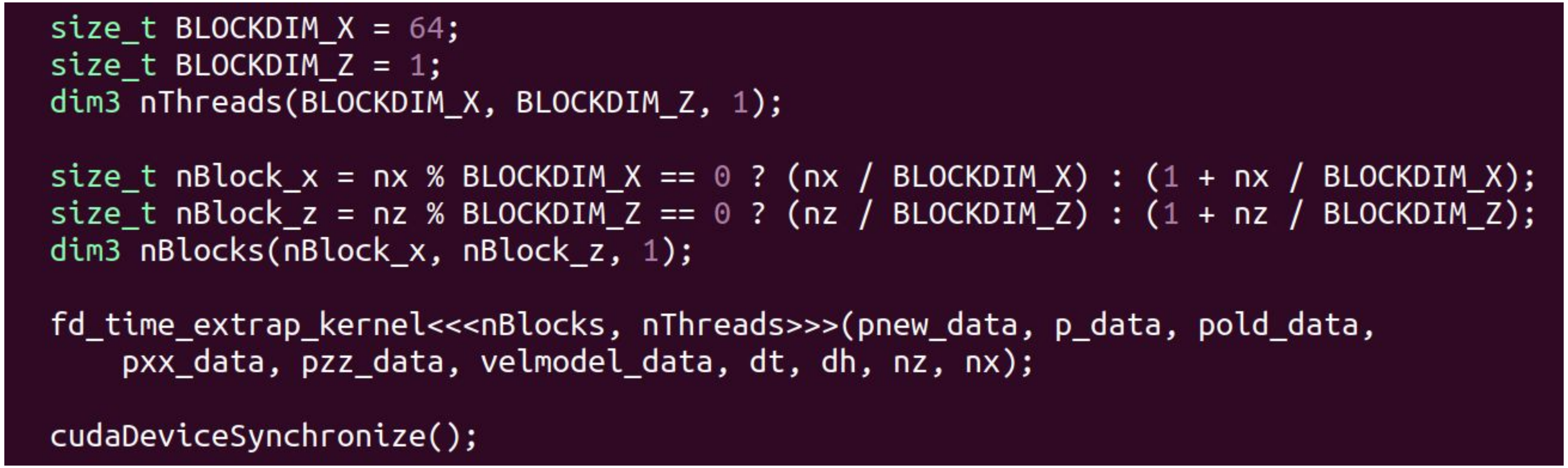
C-OpenMP implementation of time-extrap



```
#pragma omp parallel for schedule(static, 10)
for (size_t iz = 0; iz < nz; ++iz)
    for (size_t ix = 0; ix < nx; ++ix)
    {
        size_t i = iz * nx + ix;
        pnew_data[i] = (2 * p_data[i] - pold_data[i]) +
            ((dt * dt) / (dh * dh)) * (velmodel_data[i] * velmodel_data[i]) *
            (pxx_data[i] + pzz_data[i]);
    }
```

- Loop over x is innermost in order to achieve optimal sequential access pattern.
- Exploit parallelism over z in order to distribute threads across different cache lines.
- Workload is distributed in chunks of 10 rows.

C-CUDA implementation of time-extrap

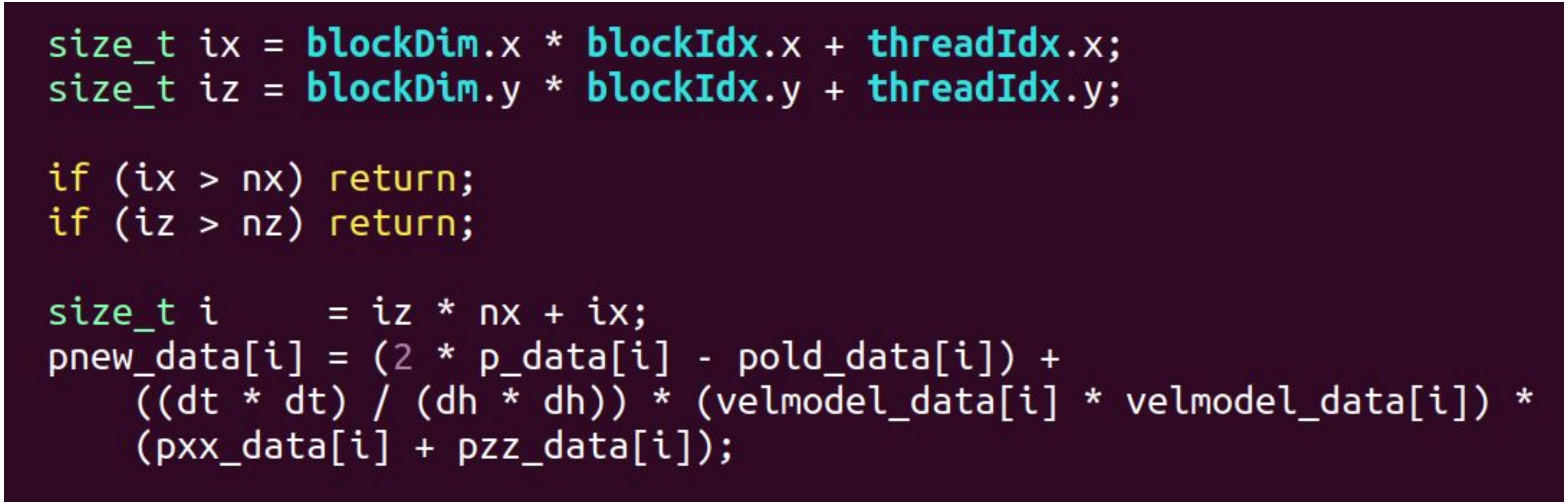


```
size_t BLOCKDIM_X = 64;
size_t BLOCKDIM_Z = 1;
dim3 nThreads(BLOCKDIM_X, BLOCKDIM_Z, 1);

size_t nBlock_x = nx % BLOCKDIM_X == 0 ? (nx / BLOCKDIM_X) : (1 + nx / BLOCKDIM_X);
size_t nBlock_z = nz % BLOCKDIM_Z == 0 ? (nz / BLOCKDIM_Z) : (1 + nz / BLOCKDIM_Z);
dim3 nBlocks(nBlock_x, nBlock_z, 1);

fd_time_extrap_kernel<<<nBlocks, nThreads>>>(pnew_data, p_data, pold_data,
    pxx_data, pzz_data, velmodel_data, dt, dh, nz, nx);

cudaDeviceSynchronize();
```

- Number of threads per block = BLOCKDIM_X * BLOCKDIM_Z * 1
- Number of blocks = nBlock_x * nBlock_z * 1
- The launch is async; for timing reasons we need to use explicit sync.

kernel in the following slide…

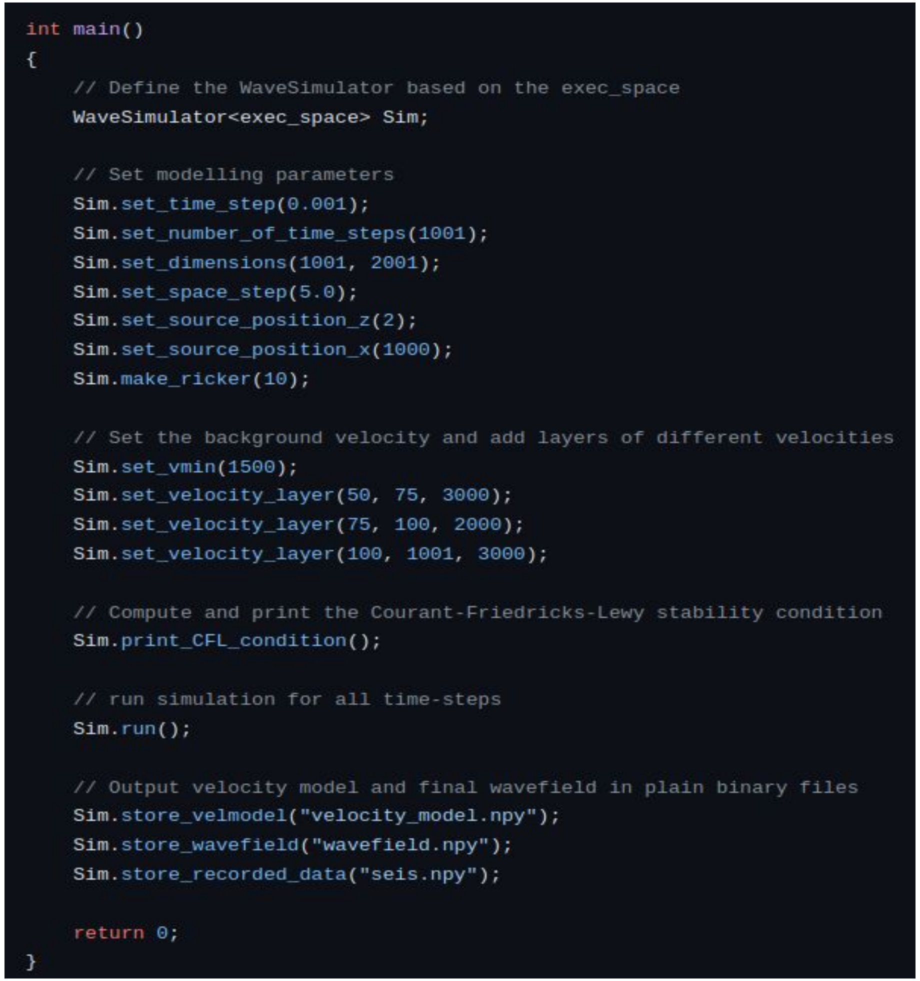
C-CUDA implementation of time-extrap



```
size_t ix = blockDim.x * blockIdx.x + threadIdx.x;
size_t iz = blockDim.y * blockIdx.y + threadIdx.y;

if (ix > nx) return;
if (iz > nz) return;

size_t i     = iz * nx + ix;
pnew_data[i] = (2 * p_data[i] - pold_data[i]) +
    ((dt * dt) / (dh * dh)) * (velmodel_data[i] * velmodel_data[i]) *
    (pxx_data[i] + pzz_data[i]);
```

- Each thread performs a single update.
- Consecutive threads access consecutive data; **ix** is aligned with the ***.x*** component of CUDA-threads.
- Avoid invalid memory access by returning with no operation the out of bound threads!!

Exercise: Modelling



```cpp
int main()
{
    // Define the WaveSimulator based on the exec_space
    WaveSimulator<exec_space> Sim;

    // Set modelling parameters
    Sim.set_time_step(0.001);
    Sim.set_number_of_time_steps(1001);
    Sim.set_dimensions(1001, 2001);
    Sim.set_space_step(5.0);
    Sim.set_source_position_z(2);
    Sim.set_source_position_x(1000);
    Sim.make_ricker(10);

    // Set the background velocity and add layers of different velocities
    Sim.set_vmin(1500);
    Sim.set_velocity_layer(50, 75, 3000);
    Sim.set_velocity_layer(75, 100, 2000);
    Sim.set_velocity_layer(100, 1001, 3000);

    // Compute and print the Courant-Friedricks-Lewy stability condition
    Sim.print_CFL_condition();

    // run simulation for all time-steps
    Sim.run();

    // Output velocity model and final wavefield in plain binary files
    Sim.store_velmodel("velocity_model.npy");
    Sim.store_wavefield("wavefield.npy");
    Sim.store_recorded_data("seis.npy");

    return 0;
}
```

The code has been developed based on C++ templates in order to hide the implementation details related to the location of data (host/device), and the selection of back-end implementation for each routine! You don't have to worry about these :)

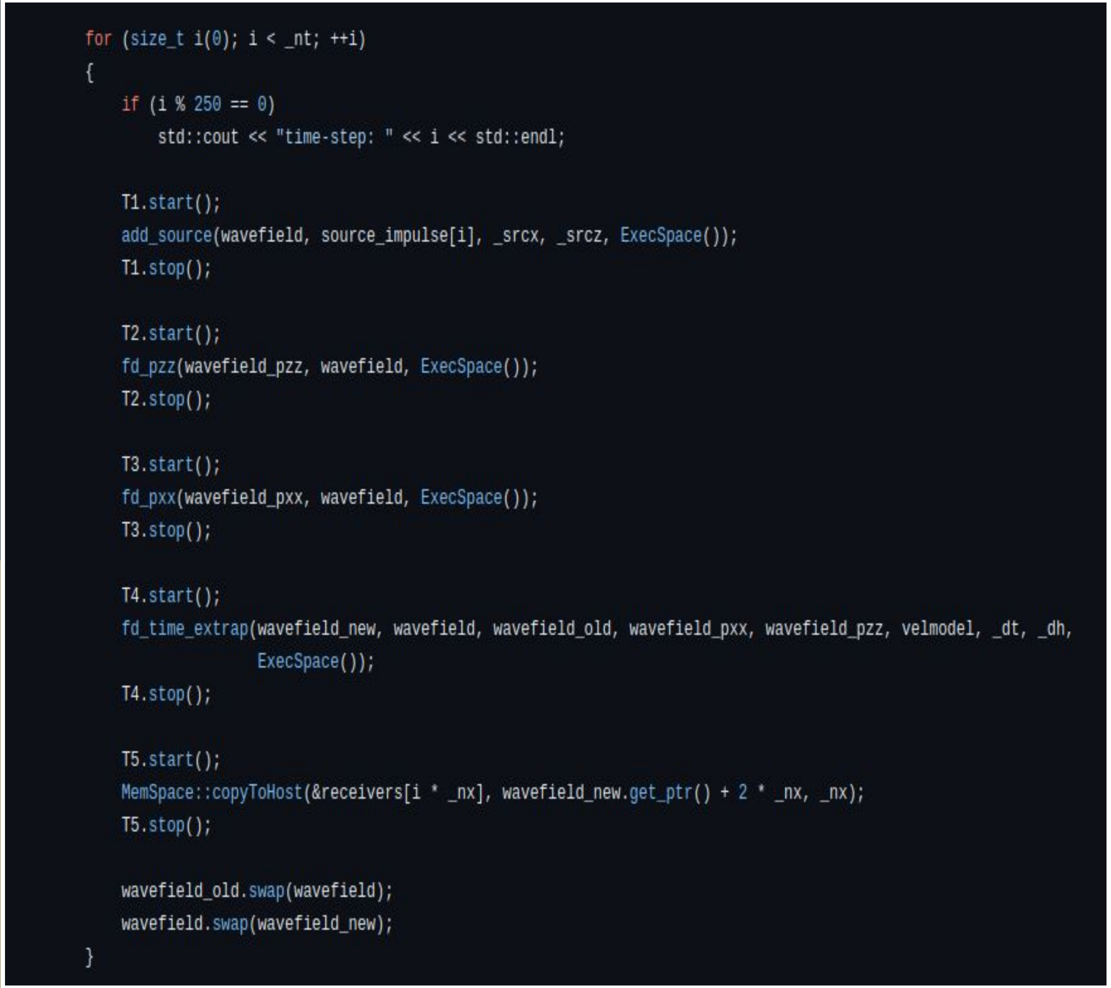
Initially, a class named `WaveSimulator` implements the algorithm, as well some utility routines that allow to modify the problem setup.

Then, the modelling parameters: dt, nt, dh, nx, nz, etc. are set

Then, we set velocity layers to create some horizontal reflectors in the velocity model.

Then, we run the simulation for all the time-steps.

Finally, we output in numpy files the velocity model, the snap-shot of wavefield at the end of simulation, the recorded seismic data at iz = 2.



```
for (size_t i(0); i < _nt; ++i)
{
    if (i % 250 == 0)
        std::cout << "time-step: " << i << std::endl;

    T1.start();
    add_source(wavefield, source_impulse[i], _srcx, _srcz, ExecSpace());
    T1.stop();

    T2.start();
    fd_pzz(wavefield_pzz, wavefield, ExecSpace());
    T2.stop();

    T3.start();
    fd_pxx(wavefield_pxx, wavefield, ExecSpace());
    T3.stop();

    T4.start();
    fd_time_extrap(wavefield_new, wavefield, wavefield_old, wavefield_pxx, wavefield_pzz, velmodel, _dt, _dh,
                   ExecSpace());
    T4.stop();

    T5.start();
    MemSpace::copyToHost(&receivers[i * _nx], wavefield_new.get_ptr() + 2 * _nx, _nx);
    T5.stop();

    wavefield_old.swap(wavefield);
    wavefield.swap(wavefield_new);
}
```

1. Add source amplitude for the current time step. (no parallelism)
2. Calculate the derivative in Z direction.
3. Calculate the derivative in X direction.
4. Calculate solution of the next time-step.
5. Save the data at depth-level iz = 2 as recorded seismic data.

*Swap the wavefields before proceeding to the next time-step.

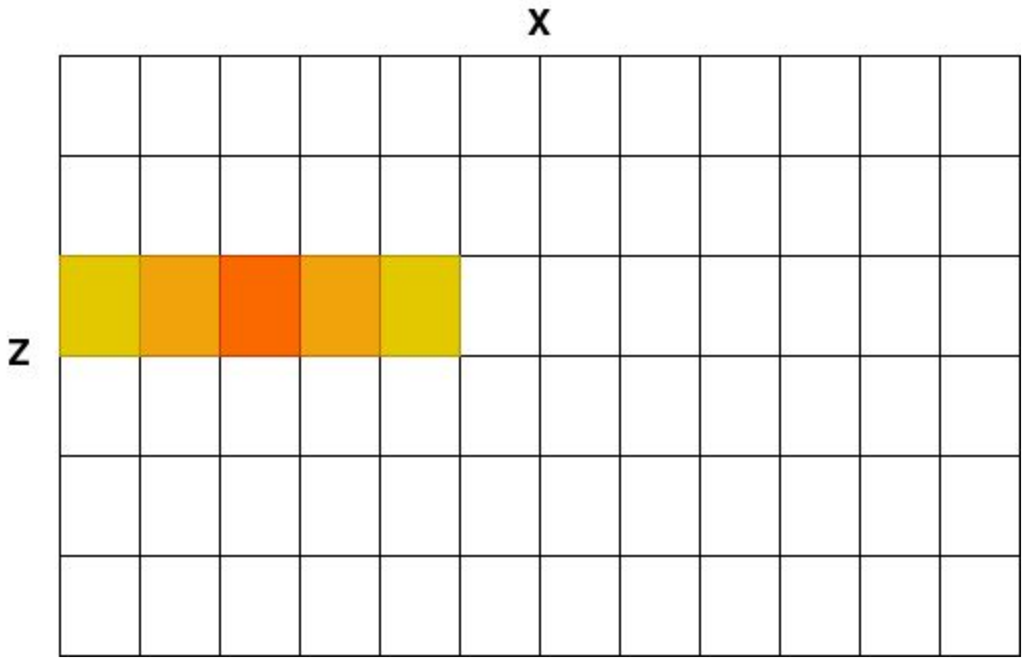
Assignment

1. Develop the two routines that implement the calculation of the derivatives over x and z using C.
2. Use C-OpenMP to parallelize workload on CPU for the two kernels that were developed in 1.
3. Do experiments using 1,2,3,..,20 threads and make a plot for the scaling. Explain the behavior.
4. What is false sharing? Implement one of the two routines such that false sharing occurs and estimate the performance degradation.
5. Use CUDA to accelerate the two routines on GPU. Report the best Blocks-Threads arrangement.

Bonus:

6. Optimization I: Fuse the two routines (Pxx & Pzz) into one. What do you expect?
7. Optimization II: Use the shared-memory on GPU to store the data reused across the threads on the same blocks. What do you expect?

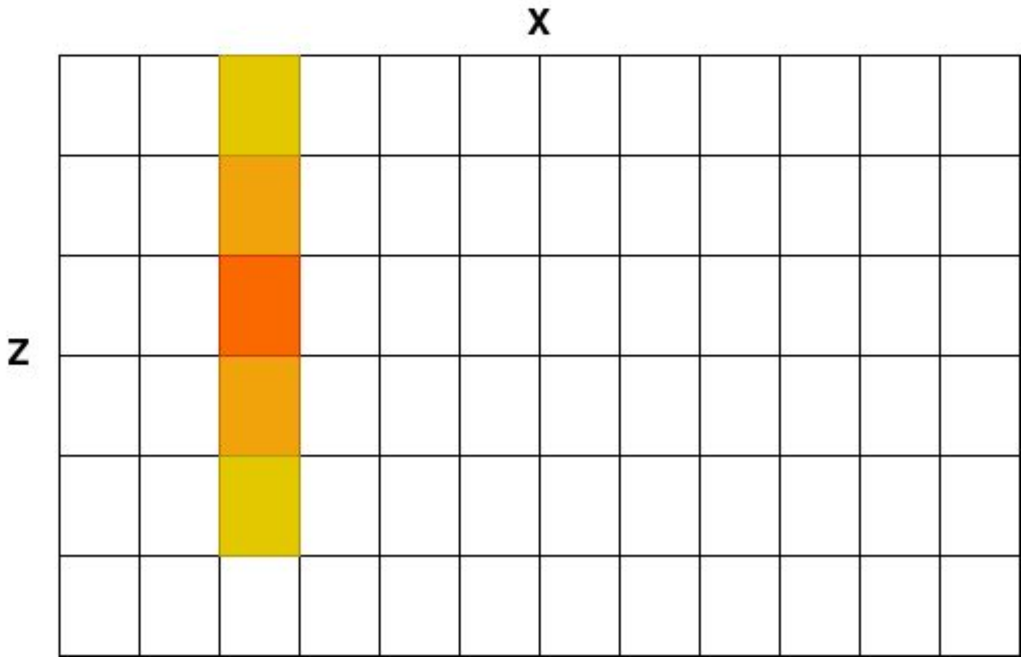
Compute: $\frac{\partial^2}{\partial x^2}$



c0 = -5/2
c1 = +4/3
c2 = -1/12

Pxx_data(z,x) = c2*P(z,x-2) + c1*P(z,x-1) + c0*P(z,x) + c1*P(z,x+1) + c2*P(z,x+2)

Compute: $\dfrac{\partial^2}{\partial z^2}$



c0 = -5/2
c1 = +4/3
c2 = -1/12

Pzz_data(z,x) = c2*P(z-2,x) + c1*P(z-1,x) + c0*P(z,x) + c1*P(z+1,x) + c2*P(z+2,x)