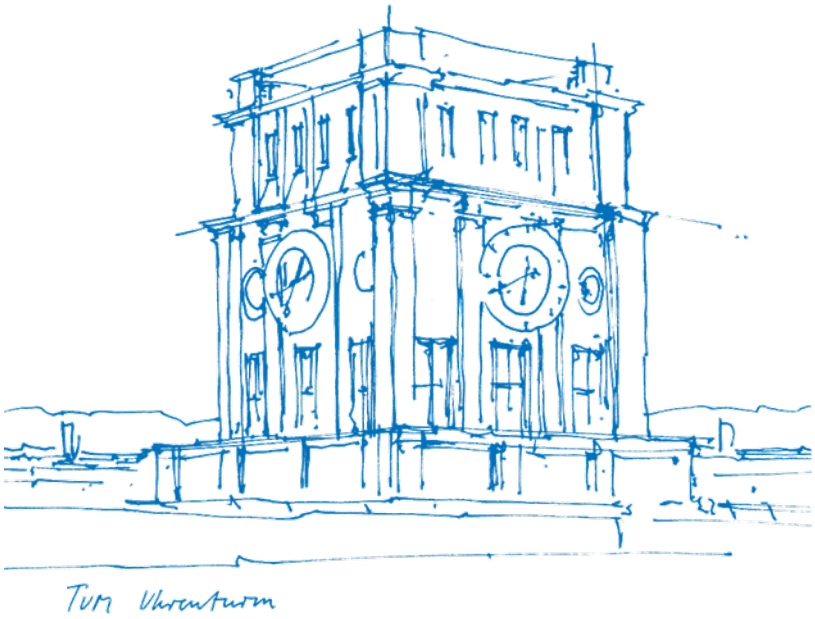
Performance-Portable Earthquake Simulations in SeisSol using SYCL

Ravil Dorozhinskii, Michael Bader
Technical University of Munich

oneAPI Workshop at LRZ: 2023

Munich, 5. June 2023



Outline

SeisSol

Software for simulating seismic waves and earthquake dynamic

based on:

- Discontinious Galerking method
- ADER time-integration scheme
- Tetrahedral meshing

supports:

- Elastic and visco-elastic Wave Propagation (WP)
- Dynamic Rupture (DR) and Point sources
- Off-fault plasticity model
- Local and Global Time Stepping schemes
- Fused-simulations

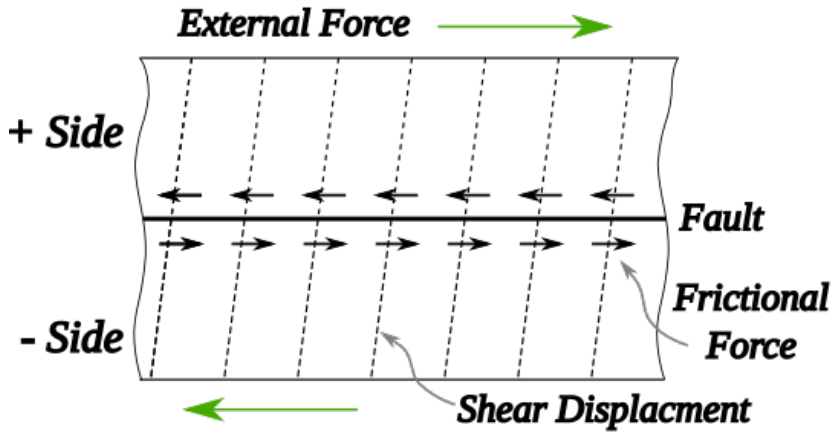
came with:

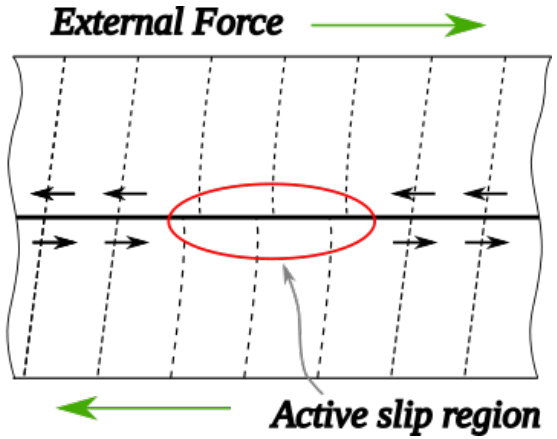
- MPI+OpenMP parallelization
- Code Generation
- GPU offloading

  WP using CUDA/HIP/SYCL

  DR using SYCL (OpenMP, experimental)

Rupture





Coulomb's model:

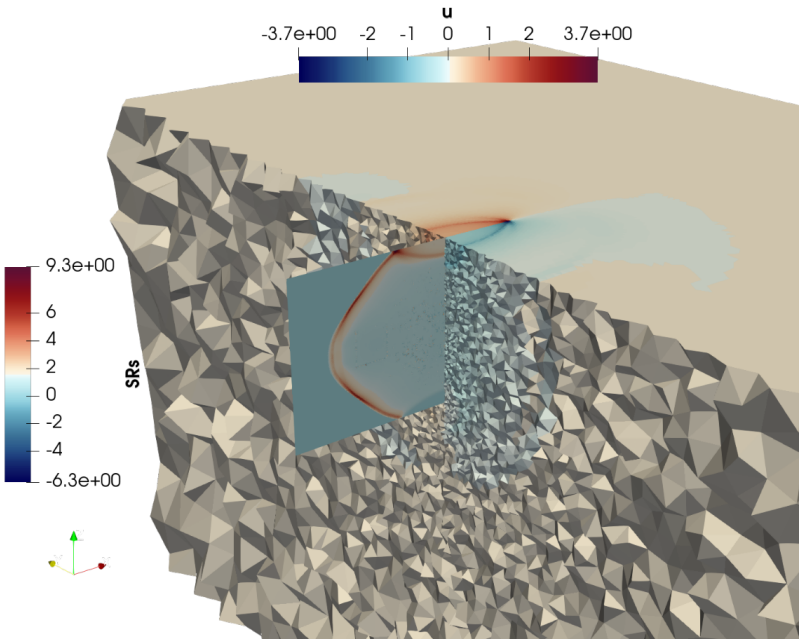
$$\tau \leq \tau_s = max(0, -\mu_f \sigma_n)$$

Slip rate:

$$V = |\Delta \mathbf{v}| = |(\mathbf{d}^+ - \mathbf{d}^-)'_t|$$

Friction modeling:

$$\mu_f = f(V, \psi)$$
$$\frac{d\psi}{dt} = g(V, \psi)$$



ADER-DG in a Nutshell

Update Scheme

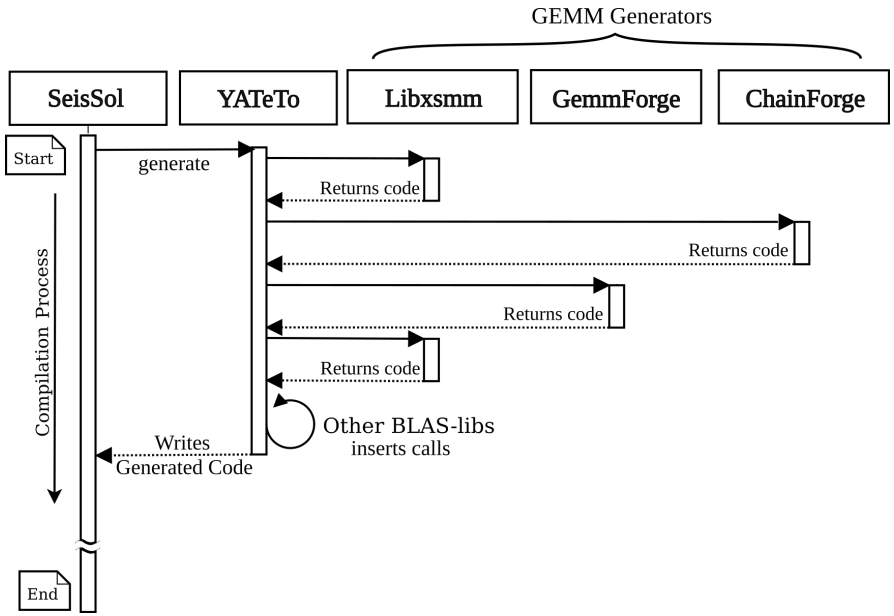
$$Q_k^{n+1} = Q_k^n + M^{-1}(K^\xi \mathscr{D}_k A_k^* + K^\eta \mathscr{D}_k B_k^* + K^\zeta \mathscr{D}_k C_k^*) \tag{1}$$
$$- \frac{1}{|J|} M^{-1} \Big(\sum_{i=1}^{4} |S_i| F^{-,i} \mathscr{D}_k \widehat{A}_k^+\Big)$$
$$- \frac{1}{|J|} M^{-1} \Big(\sum_{i=1}^{4} |S_i| F^{+,i,j_k,h_k} \mathscr{D}_{k(i)} \widehat{A}_{k(i)}^-\Big)$$

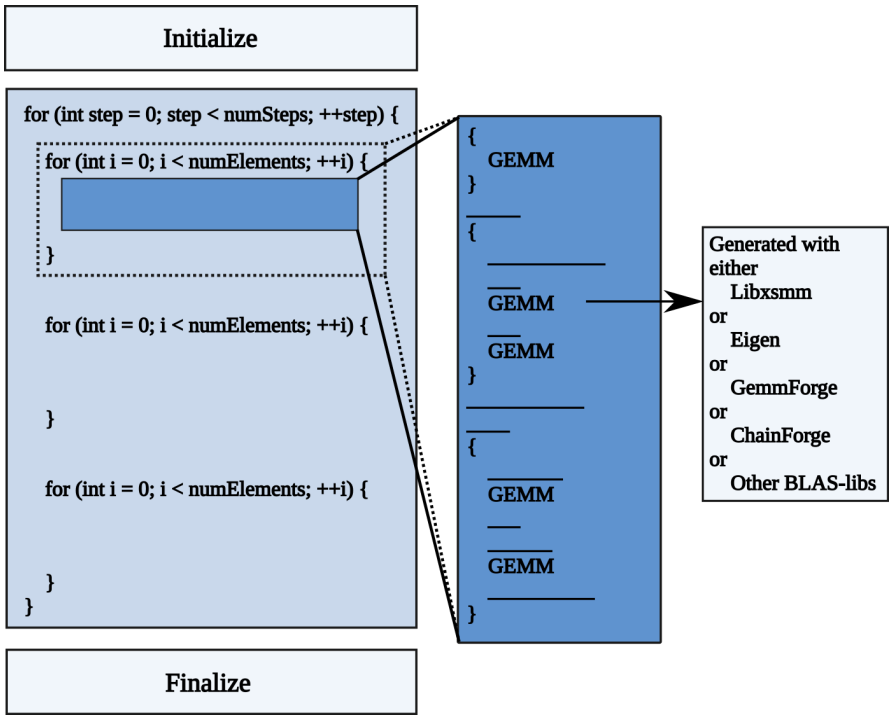
Cauchy-Kowalewski

$$\mathscr{D}_k = \sum_{j=0}^{\mathscr{O}-1} \frac{(t^{n+1} - t^n)^{j+1}}{(j+1)!} \frac{\partial^j}{\partial t^j} Q_k^n \tag{2}$$

$$\frac{\partial^{j+1}}{\partial t^{j+1}} Q_k^n = M^{-1} \left[(K^\xi)^T \Big(\frac{\partial^j}{\partial t^j} Q_k^n\Big) A_k^* + (K^\eta)^T \Big(\frac{\partial^j}{\partial t^j} Q_k^n\Big) B_k^* + (K^\zeta)^T \Big(\frac{\partial^j}{\partial t^j} Q_k^n\Big) C_k^* \right] \tag{3}$$

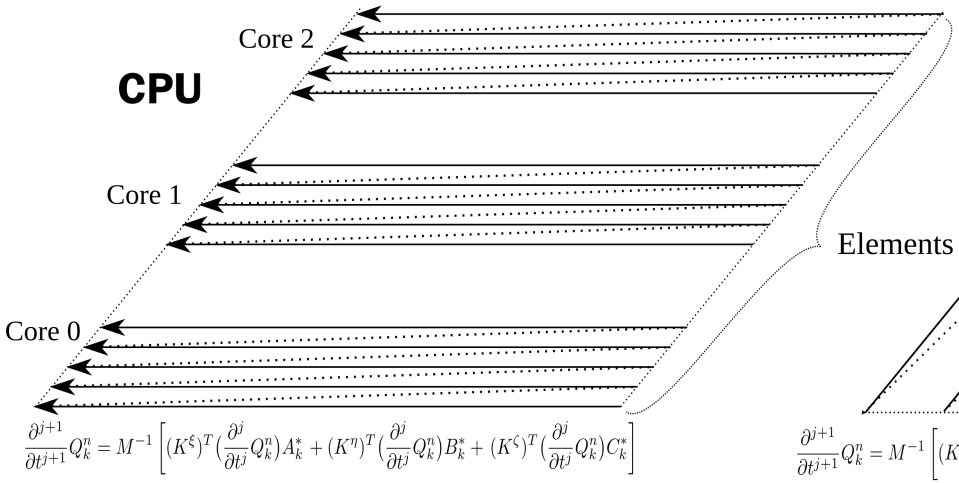
Code Structure and Generation with YATeTo

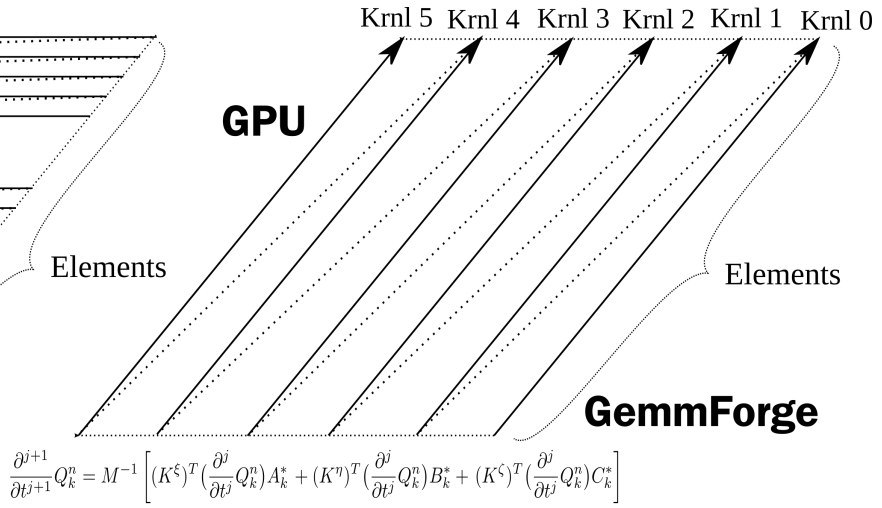


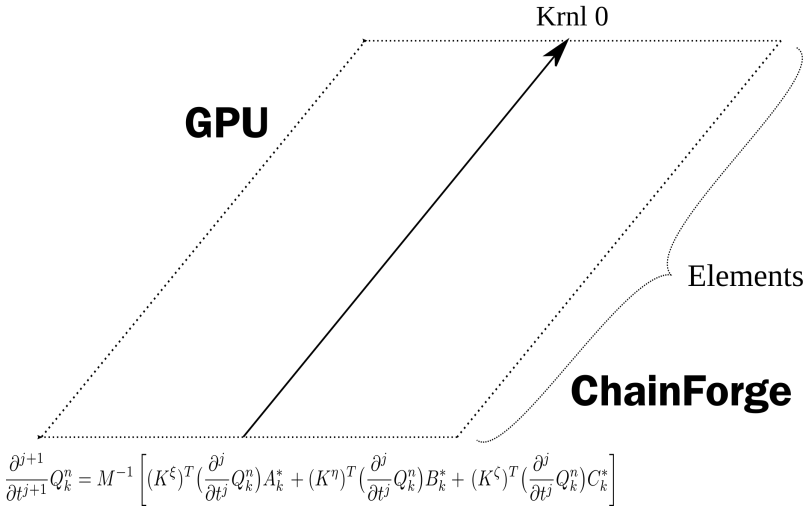


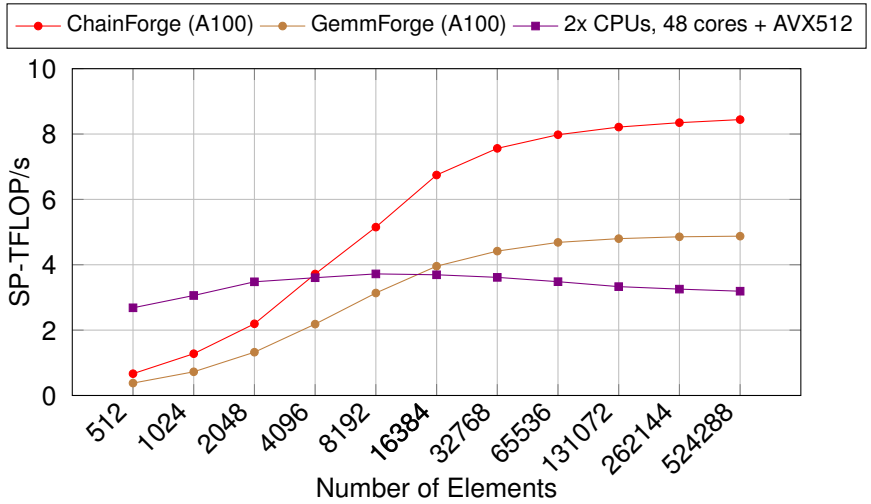

```
volumeSum = self.Q['kp']
for i in range(3):
volumeSum += self.db.kDivM[i][self.t('kl')] * self.I['lq'] * self.starMatrix(i)['qp']

volume = (self.Q['kp'] <= volumeSum)
generator.add('volume', volume)
```

GPU computing in SeisSol









Single GPU performance - SeisSol-proxy

- Obtained with *GemmForge* - Binary Batched GEMMs

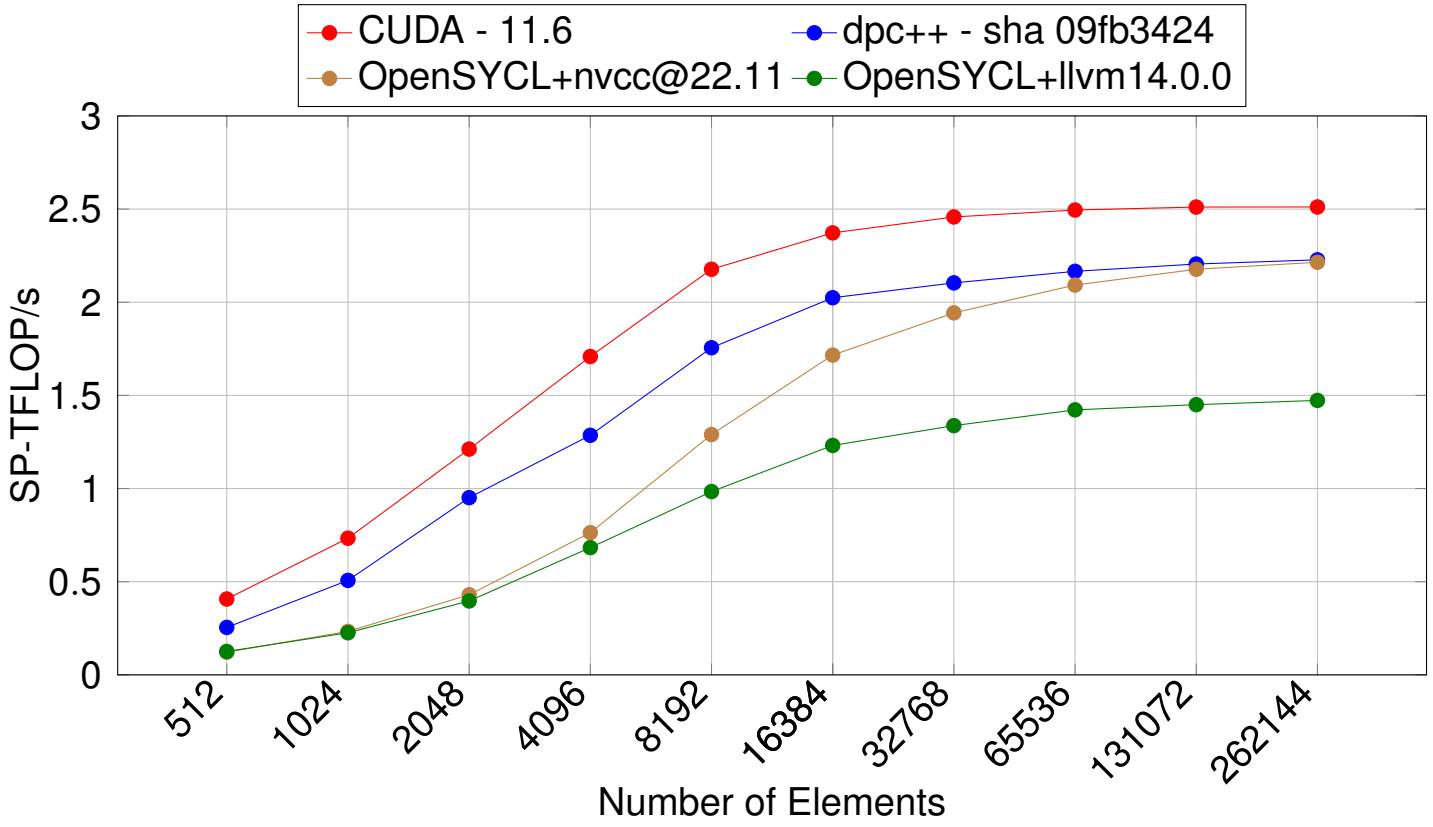


Figure: RTX 3080 Turbo

LTS in a Nutshell

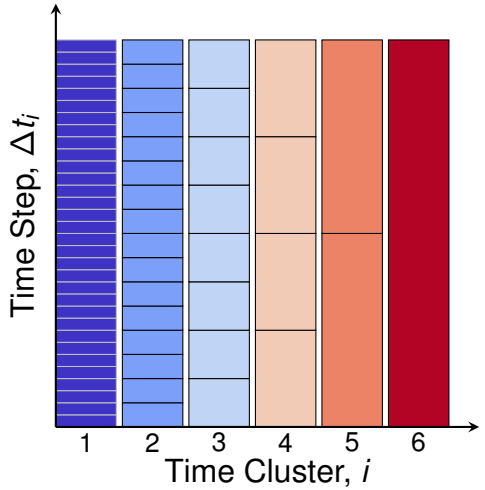


Figure. Cluster-wise time stepping ($R = 2$).

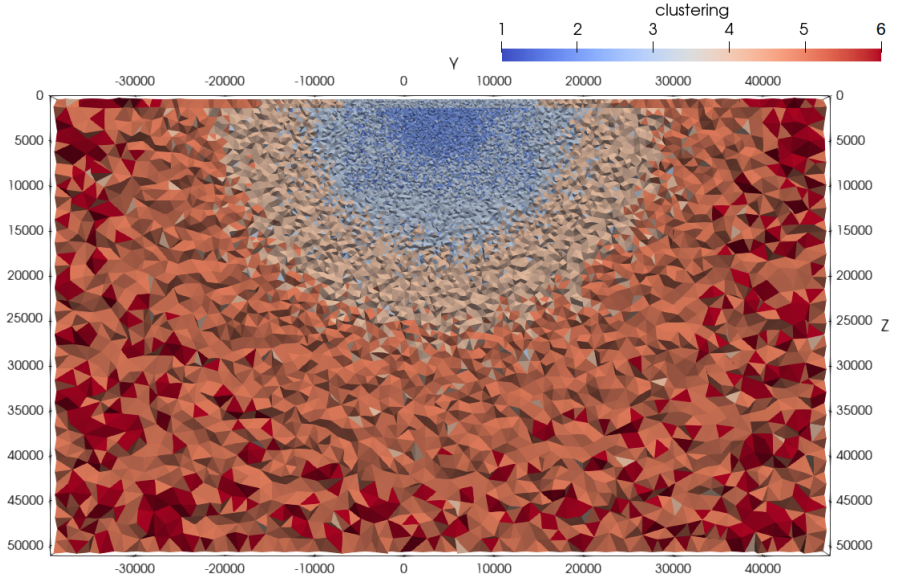


Figure. Example of elements distribution over 6 LTS clusters. A cross section of LOH.1 benchmark.

Courant-Friedrichs-Lewy condition:

- necessary condition for convergence
- determined by local wave speed and element size

Workload per element, proposed by Breuer, Heinecke, and Bader in [1]:

$$w_k = R^{L - l_k} \tag{4}$$

where R is update cluster ratio, L is the total number of clusters and l_k is a linear index of the time cluster to which element k belongs.

Graph-based Execution & LTS

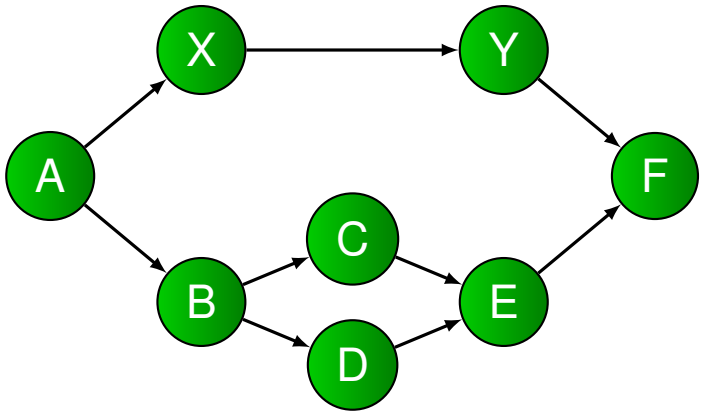


Figure. Example of a CUDA graph.

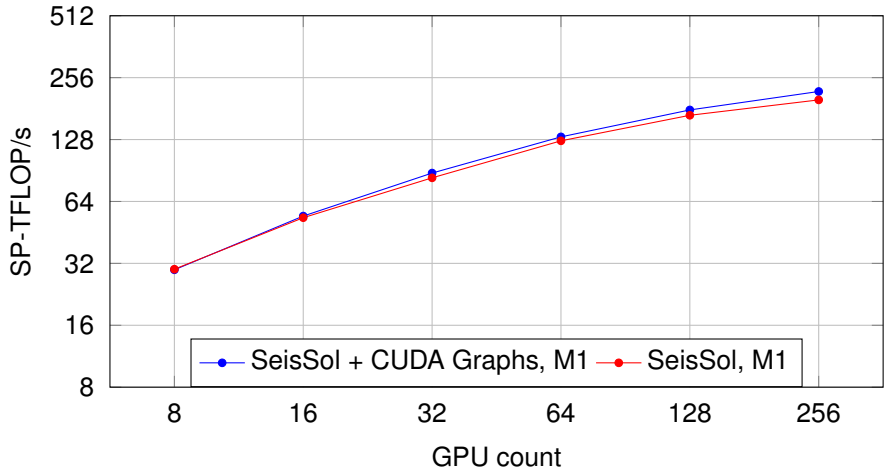


Figure. Strong scaling of LOH.1 with 5mio mesh on A100 GPU.

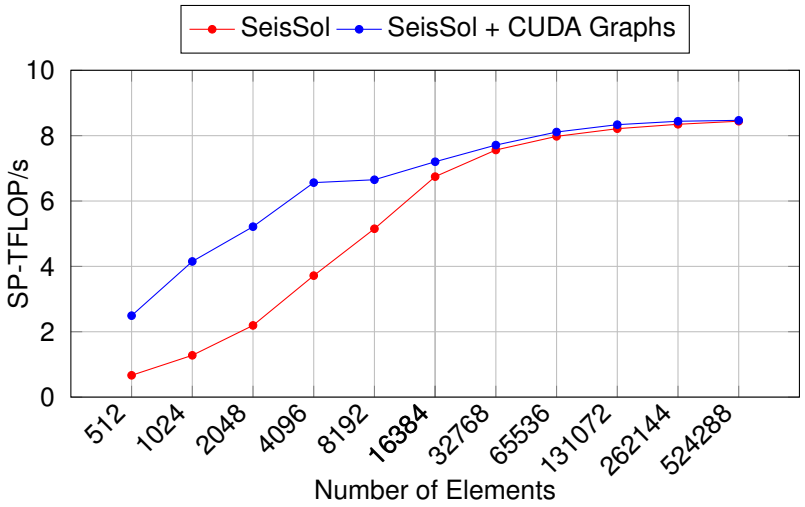


Figure. SeisSol Proxy on Nvidia A100.

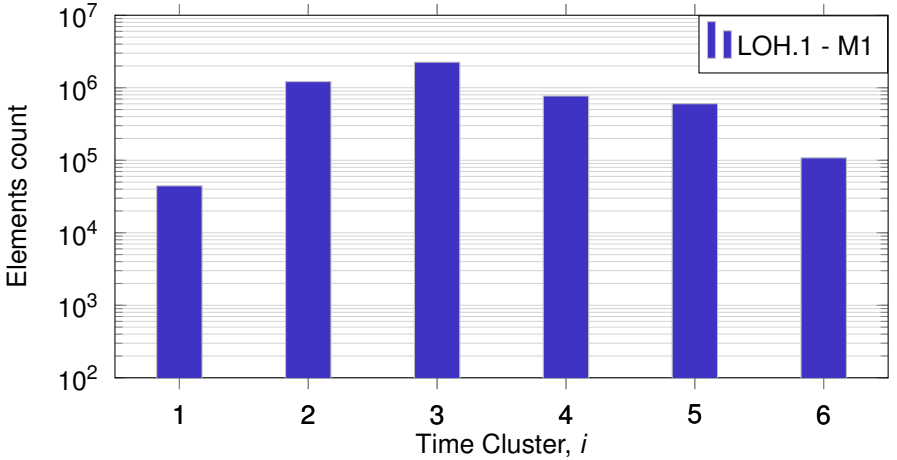
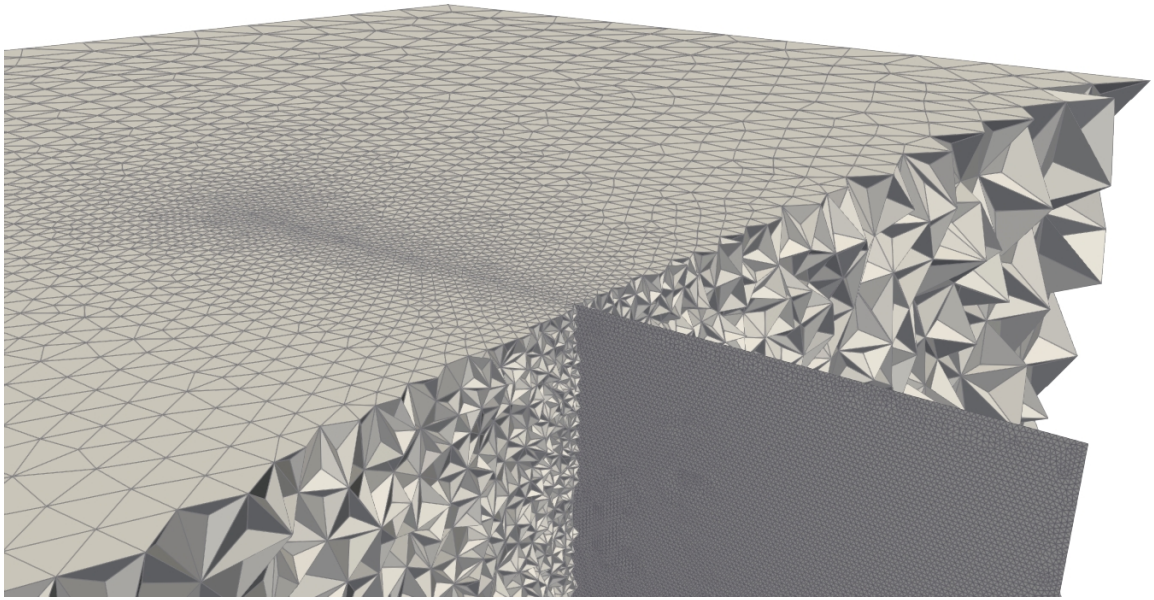
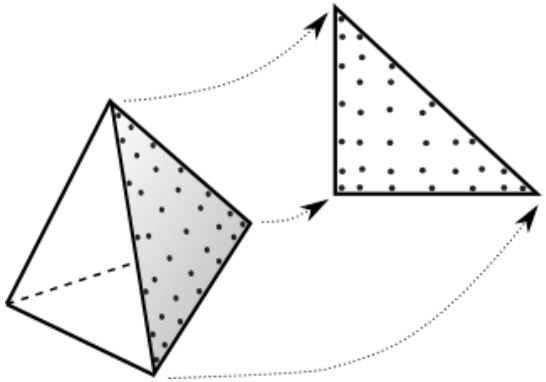


Figure. Distribution of 5mio elements of LOH.1 benchmark.

Dynamic Rupture





$$F_{pk} = A_{pr} \int_t^{t+\Delta t} \int_S \Phi_k \tilde{Q}_r \, dS \, dt$$

$$F_{pk} = A_{pr} \sum_{i=1}^{2N+1} \sum_{l=1}^{N} \omega_i^S \omega_l^T \Phi_k(\vec{\xi}_i) \tilde{Q}_{r,il}$$

Aging Law:

$$\mu_f = a \cdot \sinh^{-1}\left[\frac{V}{2V_0} exp\left(\frac{f_0 + b\ln(V_0\psi/L)}{a}\right)\right]$$

$$\frac{d\psi}{dt} = 1 - \frac{V\psi}{L}$$

$$\tau_{i,rl} \leq \tau_{s_{i,rl}} = \max(0, -\mu_f \sigma_{n_{i,rl}})$$

DR GPU offloading: SYCL

```
for (size_t timeIndex{0}; timeIndex < misc::numTimeWeights; ++timeIndex) {
  sycl::nd_range rng{{layerSize * misc::numPaddedPoints}, {misc::numPaddedPoints}};

  /*code*/

  this->queue.submit([&](sycl::handler& cgh) {
    sycl::accessor<real, 1, mode::read_write, access::target::local> buffer;

    cgh.parallel_for(rng, [=](sycl::nd_item<1> item) {
      /* kernel: part I */

      item.barrier(access::fence_space::local_space);

      /* kernel: part II */
    }

    /*code*/
  }
}
```

DR GPU offloading: OpenMP 4.5

```
for (size_t timeIndex{0}; timeIndex < misc::numTimeWeights; ++timeIndex) {
  /*code*/
  real buffer[misc::numPaddedPoints]{};

  #pragma omp target teams distribute num_teams(layerSize) \
  is_device_ptr(/*a list of pointers*/) private(buffer)       \
  device(this->deviceId) depend(inout: this->ompTag) nowait
  for (unsigned ltsFace = 0; ltsFace < layerSize; ++ltsFace) {

    #pragma omp parallel for schedule(static,1) shared(buffer)
    for (auto point = 0; point < misc::numPaddedPoints; ++point) {
      /* kernel: part I */
    }

    #pragma omp parallel for schedule(static,1) shared(buffer)
    for (auto point = 0; point < misc::numPaddedPoints; ++point) {
      /* kernel: part II */
    }
  }
  /*code*/
}
```

TPV5 Benchmark

Table: Testing Scenario

Name	TPV5
Convergence Order	6
Number Cells	595128
Number DR Cells	11794
Floating Point Format	single
Simulation Time, sec	1.0

Table: Tested Platform

Hardware	
GPU	Nvidia RTX 3080 10GB
CPU	AMD EPYC 7402 24-cores

Table: Testing Scenario

	gcc@9.3.0 omp-CPU	clang@17.0 dpcpp-GPU	gcc@9.3.0 hipSYCL-GPU	nvhpc@22.11 omp@5.1-GPU	clang@14.0 omp@4.5-GPU	gcc@11.0 omp@4.5-GPU
Hardware	24 cores	1 core - 1 GPU	1 core - 1 GPU	1 core - 1 GPU	1 core - 1 GPU	1 core - 1 GPU
Performance, GFLOP/s	1316.73	3315.262	3007.45	2811.88	1632.85	525.08
Elapsed time, sec	266.742	146.881	161.927	182.985	280.405	871.99
Total kernels time, sec	264.645	120.159	121.575	122.479	271.176	862.6
YATeTo backend	LIBSXMM	ChainForge	ChainForge	ChainForge	ChainForge	ChainForge

Conclusion

OpenMP offloading is neither compiler nor performance portable at this moment!

Profiling

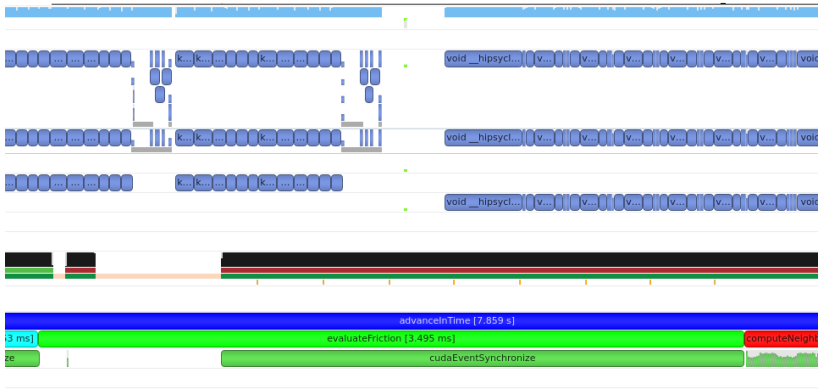


Figure: hipSYCL - llvm@14.0.0

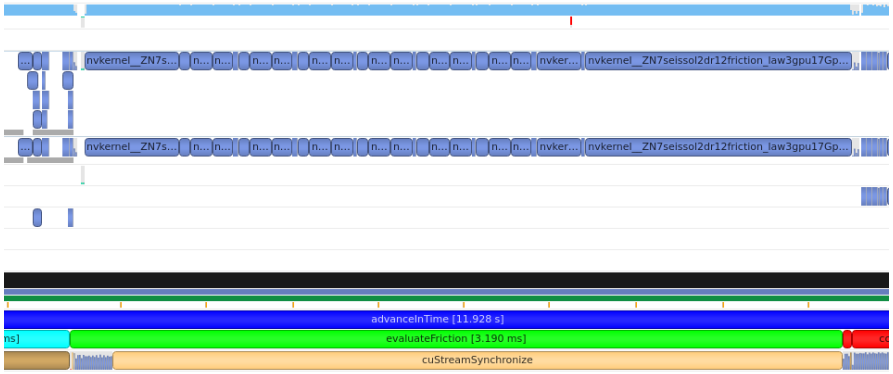


Figure: nvhpc@22.11

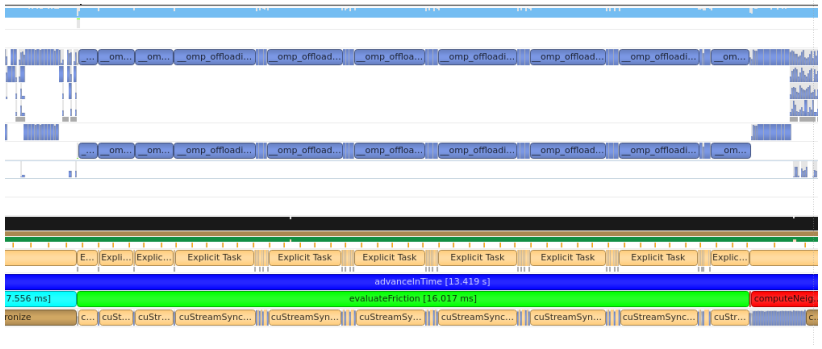
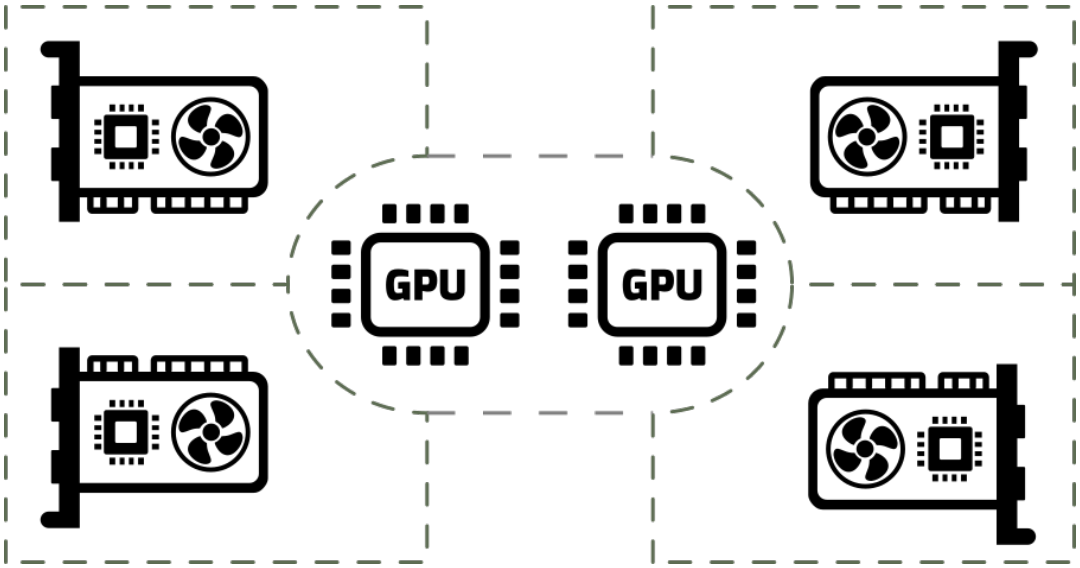


Figure: clang@14.0.0

gcc@11.3.0

MPI Process - CUDA/SYCL GPU Binding




```
#!/bin/bash

case "${SLURM_LOCALID}" in
  0) cpus=<socket-0,numa-node-0> ;;
  1) cpus=<socket-0,numa-node-1> ;;
  2) cpus=<socket-1,numa-node-0> ;;
  3) cpus=<socket-1,numa-node-1> ;;
esac

export CUDA_VISIBLE_DEVICES="${SLURM_LOCALID}"
numactl --physcpubind=$cpus "$@"
```

Conclusion

- Use vendor-specific GPU languages for the generated code (WP)
  - vendor-specific compiler backend ⇒ better code ⇒ higher performance
  - better information from vendor-specific profiling tools
  - access to advanced features (e.g., CUDA-Graphs, `cuda::memcpy_async`)
- Use SYCL for conventional scientific code (DR)
  - low enough for achieving decent performance
  - high enough for achieving less source code maintenance
- No OpenMP offloading at this moment
  - requires specific compilers versions ⇒ annoys users
  - different pragma clauses for different compilers ⇒ no source-code portability
  - “opaque” GPU kernels ⇒ difficult to find performance bottlenecks
  - prescriptive model but with a huge degree of freedom

Nice to have in the future:

- Alternative device selection
  - based of the vendor selection mechanism (using integers)
- unified SYCL CMake/Autotools interfaces