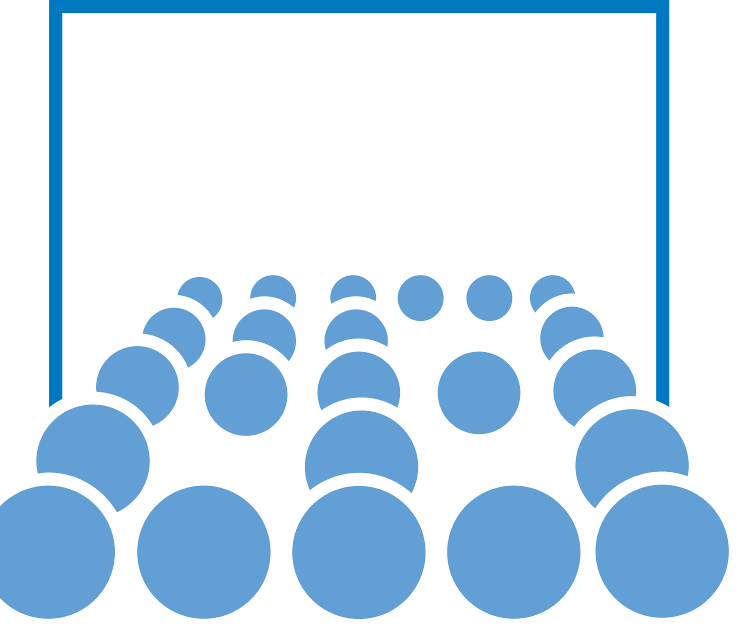
Tuning Sparse and Dense Matrix Operators in SeisSol

Alex Breuer, Alex Heinecke, S. Rettenberger,
Michael Bader, Alice Gabriel*, Christian Pelties*

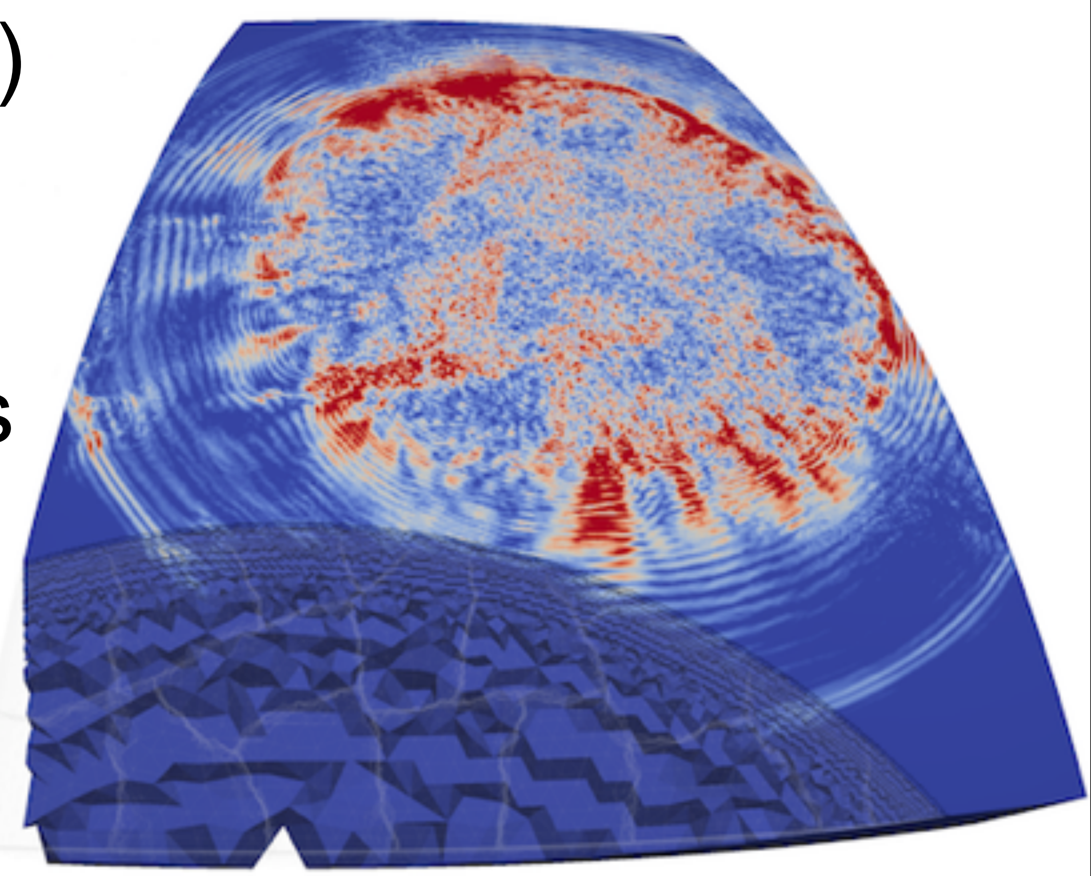
02/21/2014



*Ludwig-Maximilians-Universität München

SeisSol in a Nutshell

- Full elastic wave equations in 3D and complex heterogeneous media
- Dynamic Rupture w.o. artificial oscillations
- High order: ADER(time)-DG(space)
- Unstructured tetrahedral meshes
- Highly Optimized Compute Kernels
- Massively parallel



2009 L'Aquila, in collaboration with S. Wenk, LMU

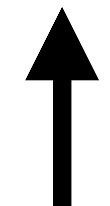
Where's the road going?

"Development of more realistic implementations of dynamic or kinematic representations of fault rupture, including simulation of higher frequencies (up to 10+ Hz)."

2013 Science Collaboration Plan
Southern California Earthquake Center

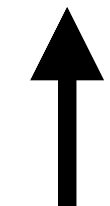
Physics To Compute Kernels

$$Q_k^{n+1} = Q_k^n - \mathcal{B}_k\left(\mathcal{J}_k^{n,n+1}, \mathcal{J}_{k(1)}^{n,n+1}, \ldots, \mathcal{J}_{k(4)}^{n,n+1}\right) + \mathcal{V}_k\left(\mathcal{J}_k^{n,n+1}\right)$$



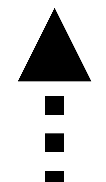
SeisSol's Compute Kernels

$$\hat{q}_b^{n+1} = \hat{q}_b^n - \frac{1}{|J|m_b}\left(\int_{t^n}^{t^{n+1}} \int_{\partial T_k} \phi_b f(q) \cdot n \, d\vec{x}dt - \int_{t^n}^{t^{n+1}} \int_{T_k} \nabla\phi_b \cdot f(q) \, d\vec{x}dt\right)$$



DG-Formulation

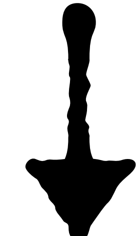
$$q_t + A(\vec{x})q_x + B(\vec{x})q_y + C(\vec{x})q_z = 0$$



Elastic Wave Equations

Time Kernel

$$Q_k^{n+1} = Q_k^n - \mathcal{B}_k\left(\mathcal{J}_k^{n,n+1}, \mathcal{J}_{k(1)}^{n,n+1}, \dots, \mathcal{J}_{k(4)}^{n,n+1}\right) + \mathcal{V}_k\left(\mathcal{J}_k^{n,n+1}\right)$$



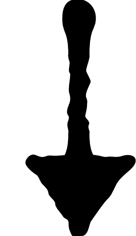
Recursive Scheme

$$\mathcal{J}_k^{n,n+1} := \mathcal{J}_k(t^n, t^{n+1}, Q_k^n) = \sum_{j=0}^{\mathcal{O}-1} \frac{(t^{n+1} - t^n)^{j+1}}{(j+1)!} \frac{\partial^j}{\partial t^j} Q_k(t^n)$$

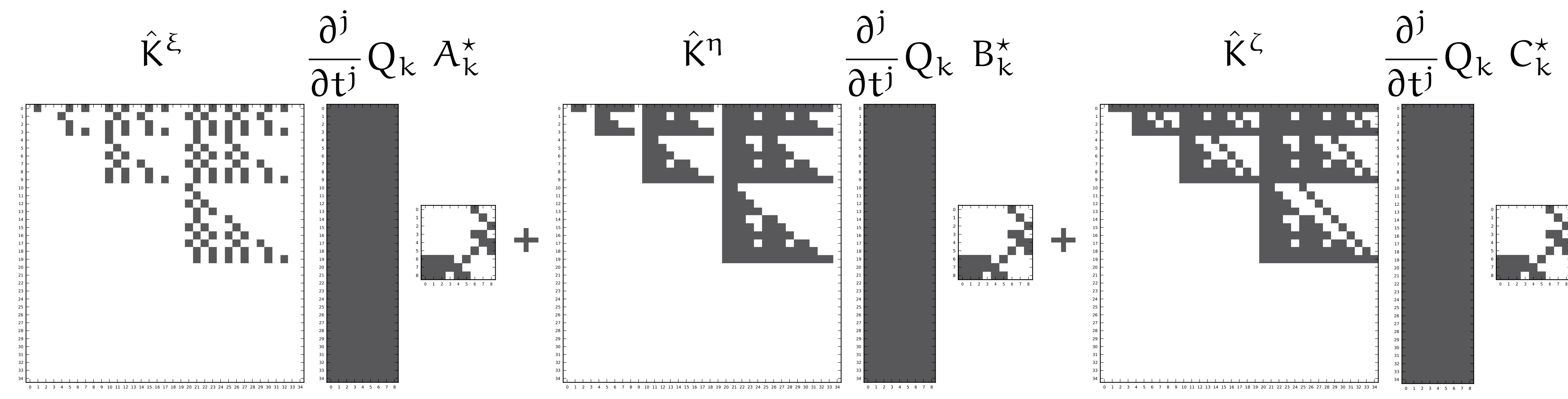
$$\frac{\partial^{j+1}}{\partial t^{j+1}} Q_k = -\hat{K}^\xi \left(\frac{\partial^j}{\partial t^j} Q_k\right) A_k^\star - \hat{K}^\eta \left(\frac{\partial^j}{\partial t^j} Q_k\right) B_k^\star - \hat{K}^\zeta \left(\frac{\partial^j}{\partial t^j} Q_k\right) C_k^\star$$

Sparsity: Time Kernel

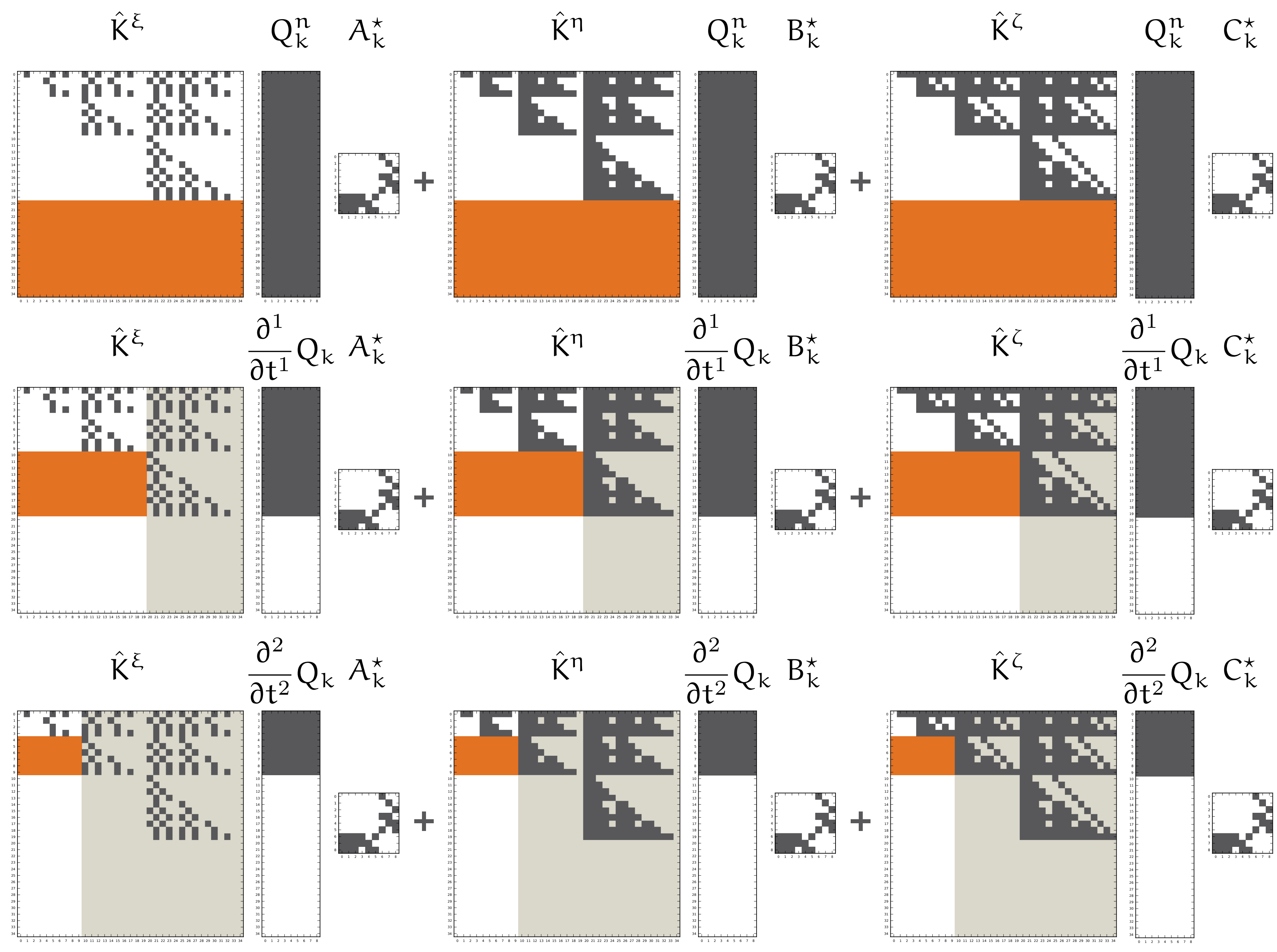
$$Q_k^{n+1} = Q_k^n - \mathcal{B}_k\left(\mathcal{J}_k^{n,n+1}, \mathcal{J}_{k(1)}^{n,n+1}, \ldots, \mathcal{J}_{k(4)}^{n,n+1}\right) + \mathcal{V}_k\left(\mathcal{J}_k^{n,n+1}\right)$$



Recursive Scheme

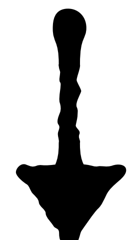


Recursive Time Kernel

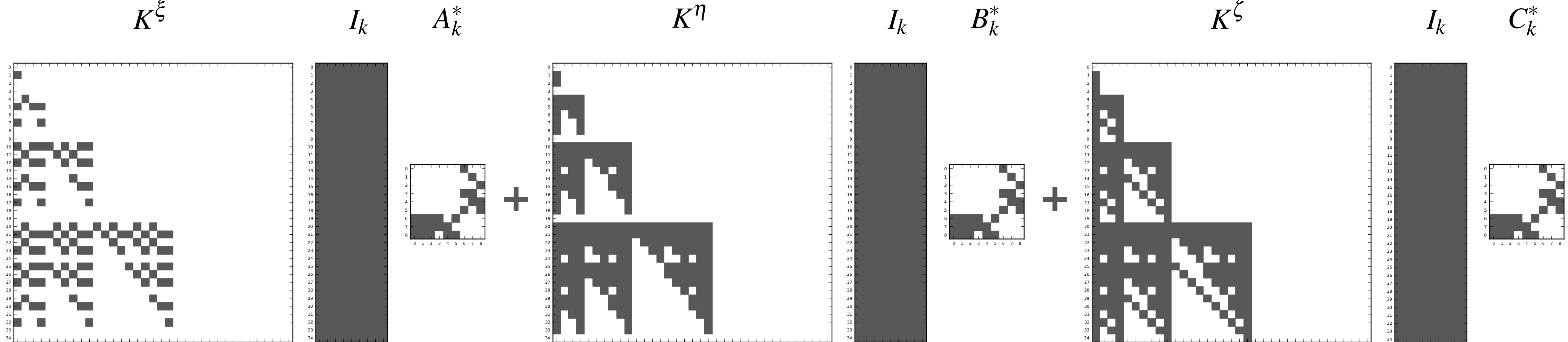


Volume Kernel

$$Q_k^{n+1} = Q_k^n - \mathcal{B}_k\left(\mathcal{J}_k^{n,n+1}, \mathcal{J}_{k(1)}^{n,n+1}, \ldots, \mathcal{J}_{k(4)}^{n,n+1}\right) + \boxed{\mathcal{V}_k\left(\mathcal{J}_k^{n,n+1}\right)}$$

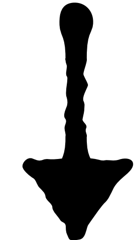


$$\boxed{\mathcal{V}_k\left(\mathcal{J}_k^{n,n+1}\right)} = \tilde{K}^{\xi}\left(\mathcal{J}_k^{n,n+1}\right)A_k^{\star} + \tilde{K}^{\eta}\left(\mathcal{J}_k^{n,n+1}\right)B_k^{\star} + \tilde{K}^{\zeta}\left(\mathcal{J}_k^{n,n+1}\right)C_k^{\star}$$



Boundary Kernel

$$Q_k^{n+1} = Q_k^n - \mathcal{B}_k\left(\mathcal{I}_k^{n,n+1}, \mathcal{I}_{k(1)}^{n,n+1}, \ldots, \mathcal{I}_{k(4)}^{n,n+1}\right) + \mathcal{V}_k\left(\mathcal{I}_k^{n,n+1}\right)$$

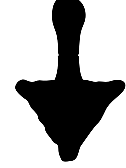


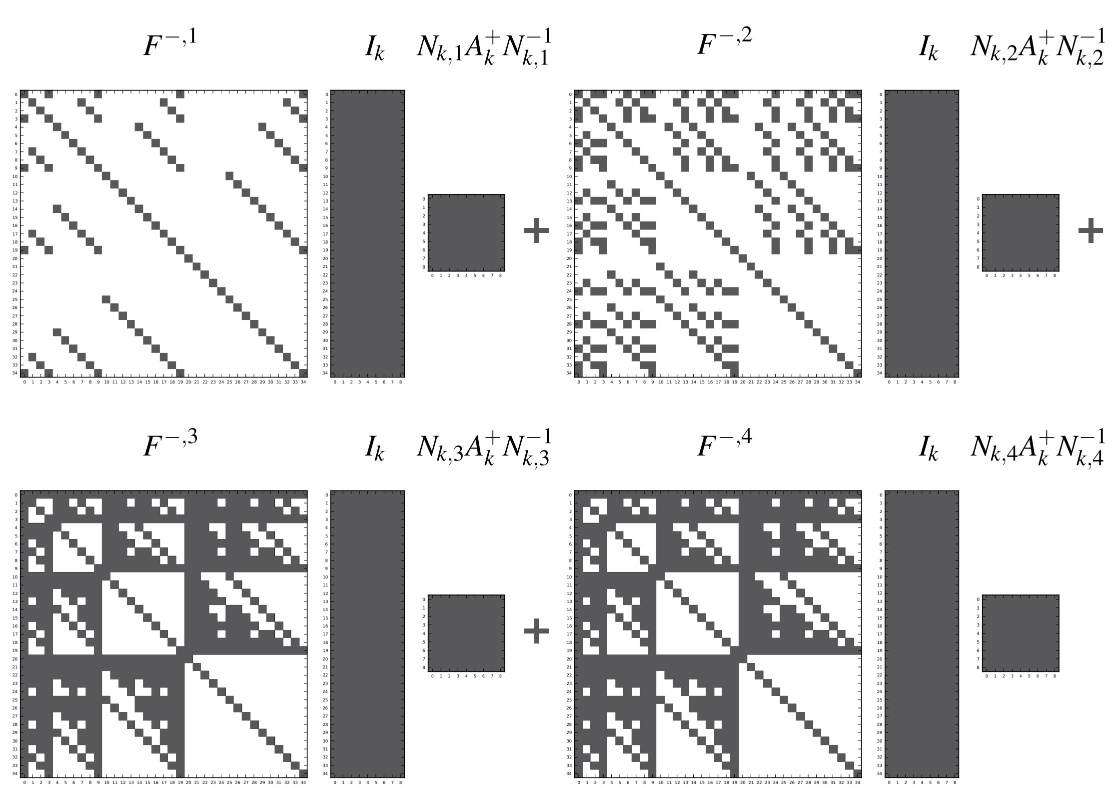
$$\mathcal{B}_k\left(\mathcal{I}_k^{n,n+1}, \mathcal{I}_{k(1)}^{n,n+1}, \ldots, \mathcal{I}_{k(4)}^{n,n+1}\right) = \sum_{i=1}^{4} \left(M^{-1}F^{-,i}\right) I_k^{n,n+1} \left(\frac{|S_k|}{|J_k|} N_{k,i} A_k^{+} N_{k,i}^{-1}\right)$$

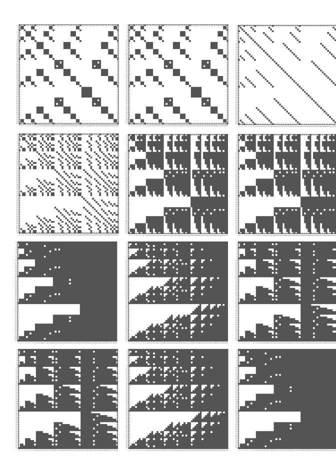
$$+ \sum_{i=1}^{4} \left(M^{-1}F^{+,i,j_k(i),h_k(i)}\right) I_{k(i)}^{n,n+1} \left(\frac{|S_k|}{|J_k|} N_{k,i} A_{k(i)}^{-} N_{k,i}^{-1}\right)$$

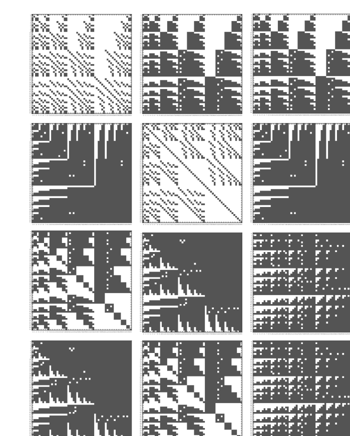
Sparsity: Boundary Kernel

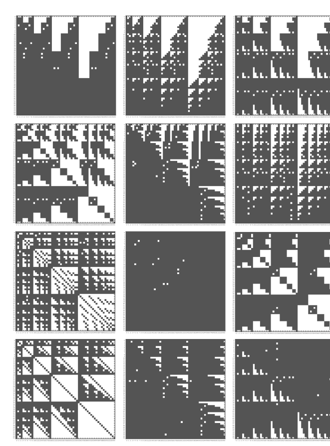
$$\mathcal{Q}_k^{n+1} = \mathcal{Q}_k^n - \mathcal{B}_k\left(\mathcal{I}_k^{n,n+1}, \mathcal{I}_{k(1)}^{n,n+1}, \ldots, \mathcal{I}_{k(4)}^{n,n+1}\right) + \mathcal{V}_k\left(\mathcal{I}_k^{n,n+1}\right)$$

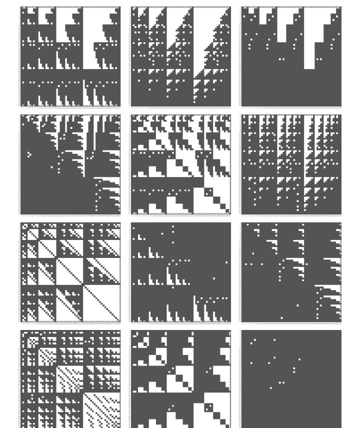












Generated Code: Dense × Sparse

```
inline void generatedMatrixMultiplication_starMatrix_3D_9_20(double* A, double* values, double* C)
{
#pragma simd
for (int i = 0; i < 20; i+=1)
{
C[(i)+(0)] += A[(i)+(120)] * values[0];
C[(i)+(0)] += A[(i)+(140)] * values[1];
C[(i)+(0)] += A[(i)+(160)] * values[2];
C[(i)+(20)] += A[(i)+(120)] * values[3];
C[(i)+(20)] += A[(i)+(140)] * values[4];
C[(i)+(20)] += A[(i)+(160)] * values[5];
C[(i)+(40)] += A[(i)+(120)] * values[6];
C[(i)+(40)] += A[(i)+(140)] * values[7];
C[(i)+(40)] += A[(i)+(160)] * values[8];
C[(i)+(60)] += A[(i)+(120)] * values[9];
C[(i)+(60)] += A[(i)+(140)] * values[10];
C[(i)+(80)] += A[(i)+(140)] * values[11];
C[(i)+(80)] += A[(i)+(160)] * values[12];
C[(i)+(100)] += A[(i)+(120)] * values[13];
C[(i)+(100)] += A[(i)+(160)] * values[14];
C[(i)+(120)] += A[(i)+(0)] * values[15];
C[(i)+(120)] += A[(i)+(60)] * values[16];
C[(i)+(120)] += A[(i)+(100)] * values[17];
C[(i)+(140)] += A[(i)+(20)] * values[18];
C[(i)+(140)] += A[(i)+(60)] * values[19];
C[(i)+(140)] += A[(i)+(80)] * values[20];
C[(i)+(160)] += A[(i)+(40)] * values[21];
C[(i)+(160)] += A[(i)+(80)] * values[22];
C[(i)+(160)] += A[(i)+(100)] * values[23];
}
#ifndef NDEBUG
num_flops += 960;
#endif

}
```

Generated Code: Sparse × Dense

stiffness_matrices_3d.hpp

stiffness_matrices_3d.hpp 〉 generatedMatrixMultiplication_kEta_9_20(double* values, double* B, double* C)

```
inline void generatedMatrixMultiplication_kEta_9_20(double* values, double* B, double* C)
{
#pragma nounroll
for (int i = 0; i < 9; i++)
{
#if defined(__SSE3__) || defined(__AVX256__)
#if defined(__SSE3__) && defined(__AVX256__)
__m256d b0 = _mm256_broadcast_sd(&B[(i*20)+0]);
#endif
#if defined(__SSE3__) && !defined(__AVX256__)
__m128d b0 = _mm_loaddup_pd(&B[(i*20)+0]);
#endif
__m128d c0_0 = _mm_loadu_pd(&C[(i*20)+1]);
__m128d a0_0 = _mm_loadu_pd(&values[0]);
#if defined(__SSE3__) && defined(__AVX256__)
c0_0 = _mm_add_pd(c0_0, _mm_mul_pd(a0_0, _mm256_castpd256_pd128(b0)));
#endif
#if defined(__SSE3__) && !defined(__AVX256__)
c0_0 = _mm_add_pd(c0_0, _mm_mul_pd(a0_0, b0));
#endif
_mm_storeu_pd(&C[(i*20)+1], c0_0);
#if defined(__SSE3__) && defined(__AVX256__)
__m256d c0_2 = _mm256_loadu_pd(&C[(i*20)+4]);
__m256d a0_2 = _mm256_loadu_pd(&values[2]);
c0_2 = _mm256_add_pd(c0_2, _mm256_mul_pd(a0_2, b0));
```

Generated Code: Dense × Dense

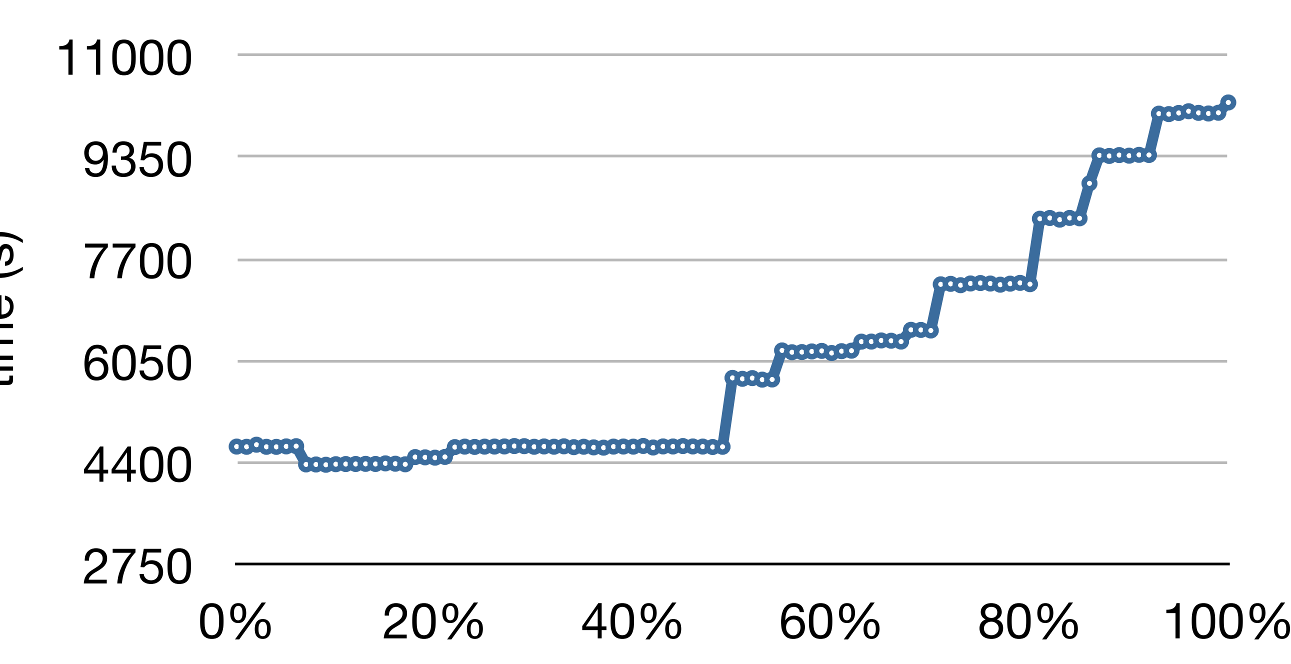
dense_matrices.hpp

dense_matrices.hpp 〉 generatedMatrixMultiplication_dense_56_9_56(double* A, double* B, double* C, double* A_prefetch = NULL, double* B_prefetch = NULL, double* C_prefetch = NULL)

```
inline void generatedMatrixMultiplication_dense_56_9_56(double* A, double* B, double* C, double* A_prefetch = NULL, double* B_prefetch = NULL
          , double* C_prefetch = NULL) {
//[...] SSE and AVX data types, SSE optimization
#if defined(__SSE3__) && defined(__AVX__)
//[...] pointers
for(int n = 0; n < 9; n+=3) {
  for(int m = 0; m < 48; m+=12) {
    //[...] pointers and loads
    for (int k = 0; k < 56; k++)
    {
    b_0 = _mm256_broadcast_sd(b0);
    //[...] more broadcasts and arithmetics on b
    a_0 = _mm256_load_pd(a0);
    a0+=4;
    c_0_0 = _mm256_add_pd(c_0_0, _mm256_mul_pd(a_0, b_0));
    c_0_1 = _mm256_add_pd(c_0_1, _mm256_mul_pd(a_0, b_1));
    c_0_2 = _mm256_add_pd(c_0_2, _mm256_mul_pd(a_0, b_2));
//[...] remaining kernel
#endif

#if defined(__MIC__)
//[...] MIC specifics
for(int n = 0; n < 9; n+=3) {
  //[...] MIC specififics
  #pragma prefetch b0,b1,b2,a0
  for(int k = 0; k < 56; k++)
  {
    //[...] MIC specifics
    c_0_0 = _mm512_fmadd_pd(a_0, b_0, c_0_0);
    c_0_1 = _mm512_fmadd_pd(a_0, b_1, c_0_1);
    c_0_2 = _mm512_fmadd_pd(a_0, b_2, c_0_2);
    a0 += 8;

//[...] remaining MIC kernel
#endif

#if !defined(__SSE3__) && !defined(__AVX__) && !defined(__MIC__)
//[...] fallback code
#endif

#ifndef NDEBUG
num_flops += 56448;
```

Sparse Dense Switch

Runtime tuning runs:
Best routine for every operator

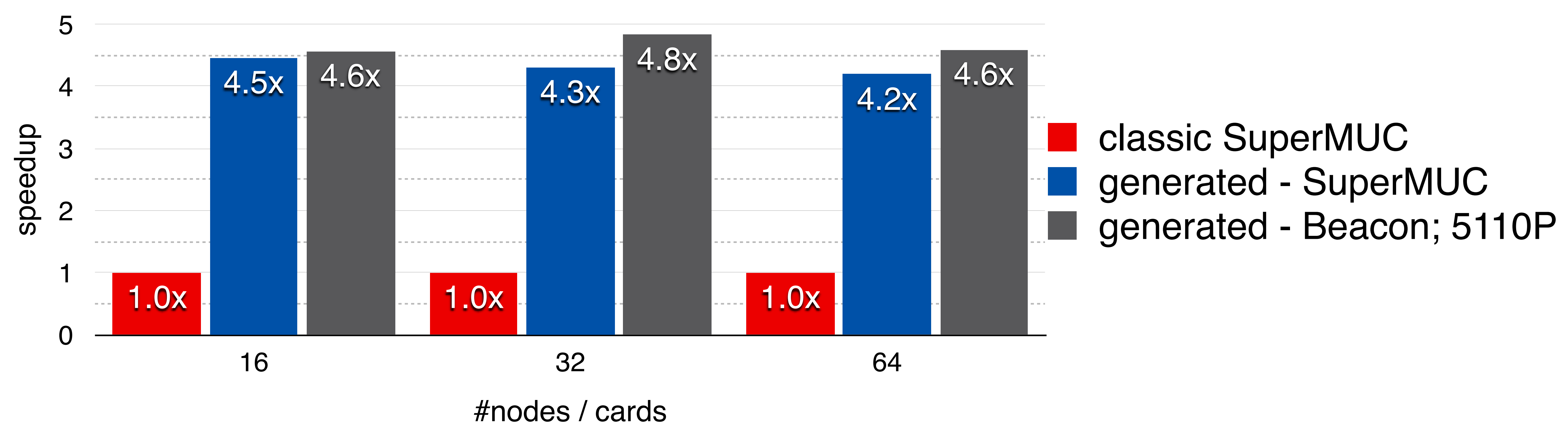


Decision matrix for all orders and operators

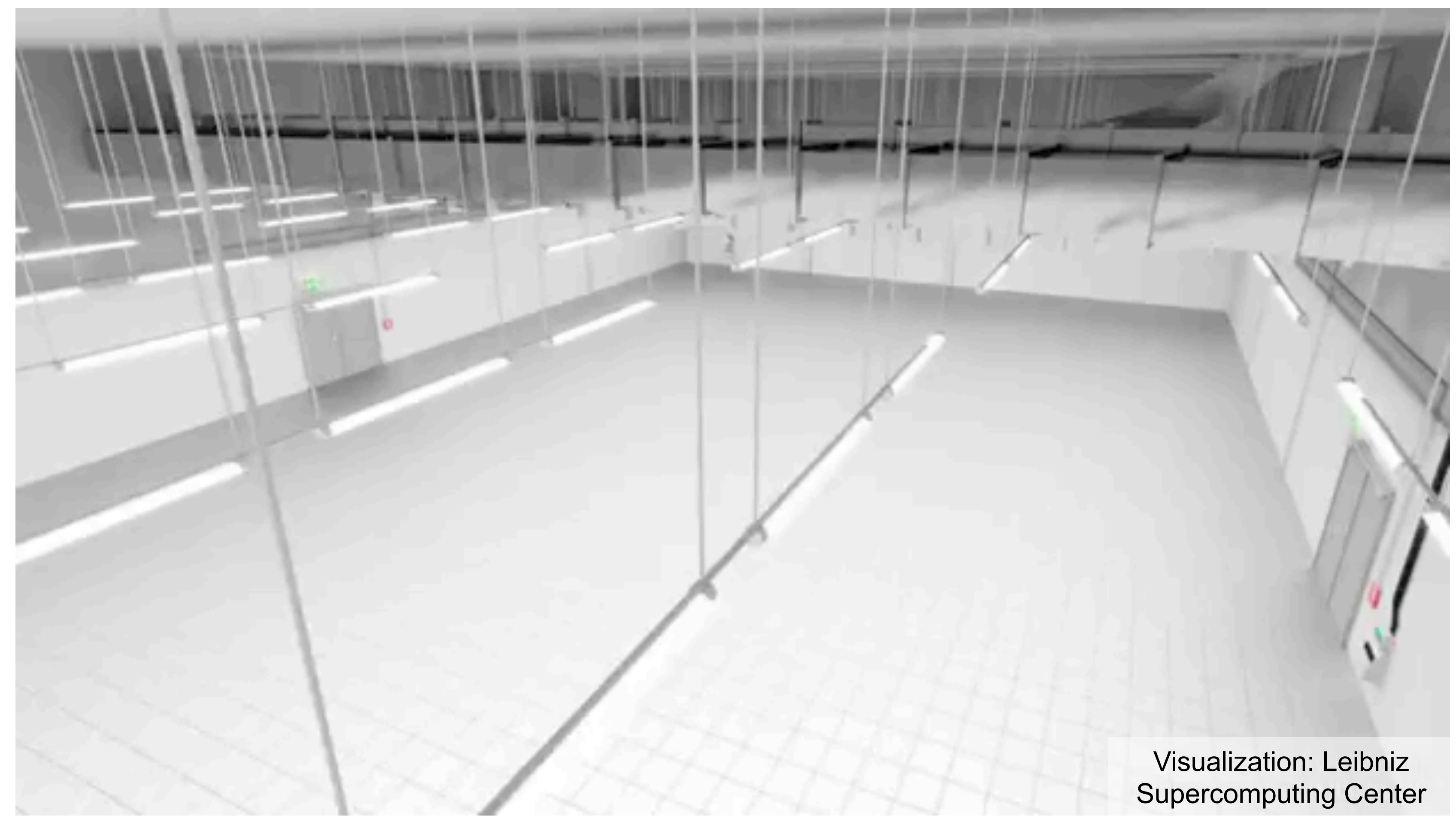
order	$\hat{K}^{\xi}$	$\hat{K}^{\xi}$	$\hat{K}^{\eta}$	$\tilde{K}^{\xi}$	$\tilde{K}^{\eta}$	$\tilde{K}^{\zeta}$	boundary
2	sparse	sparse	sparse	sparse	sparse	sparse	13%
3	sparse	sparse	dense	sparse	sparse	sparse	26%
4	sparse	sparse	dense	sparse	dense	dense	17%
5	sparse	sparse	sparse	sparse	sparse	sparse	23%
6	sparse	dense	dense	sparse	dense	dense	9%

Speedup

- SCEC LOH.1, 7,252,482 elements, 100 time steps
- 6th order in space and time (for all upcoming results)
- 100 time steps



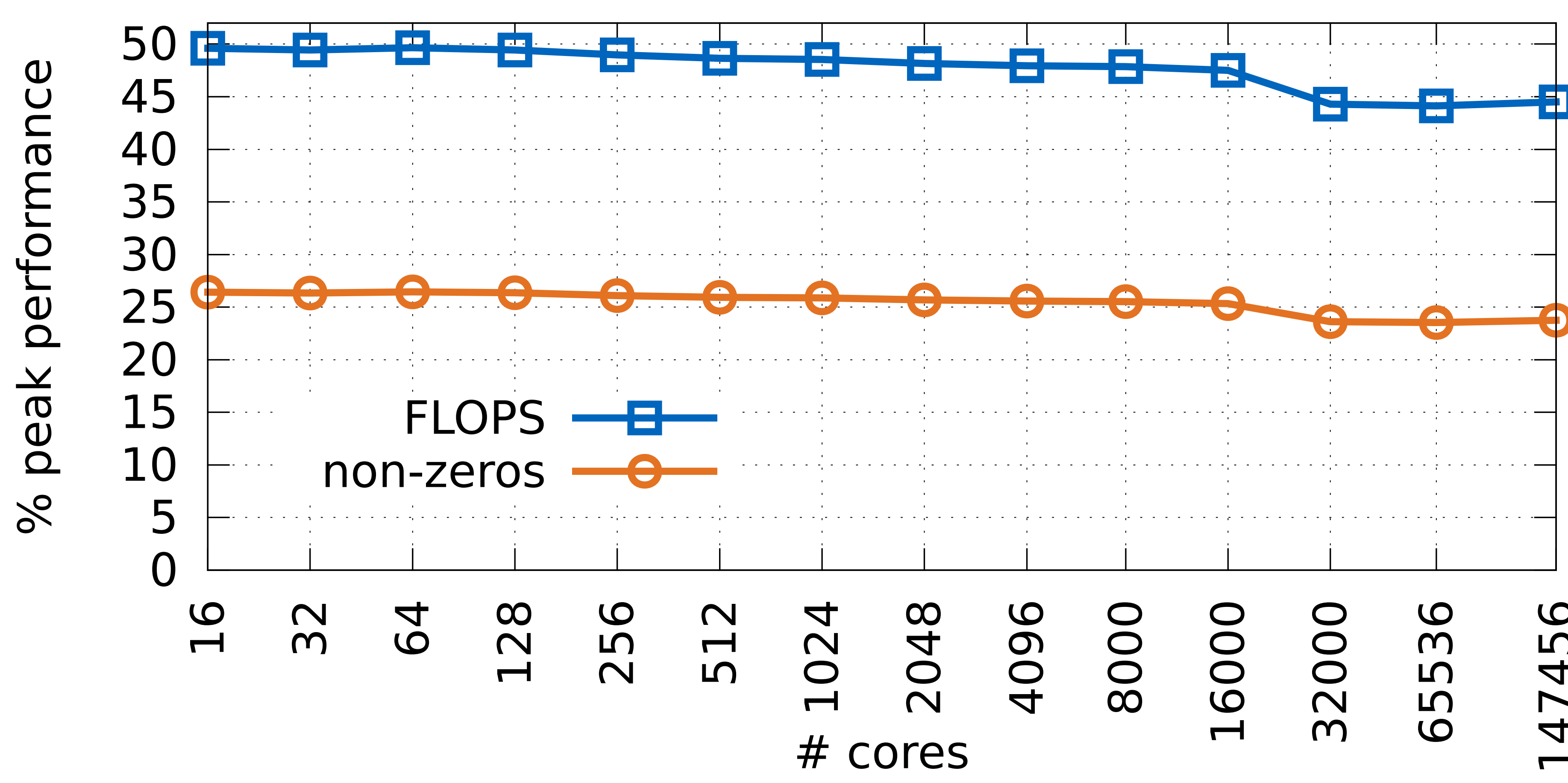
Large Scale: SuperMUC



- 3 PFLOPS (#10 in the Top500) @ LRZ, Germany
- 147K SNB-cores
- Islands: 8K cores, 4:1 pruned tree (FDR-10 IB)

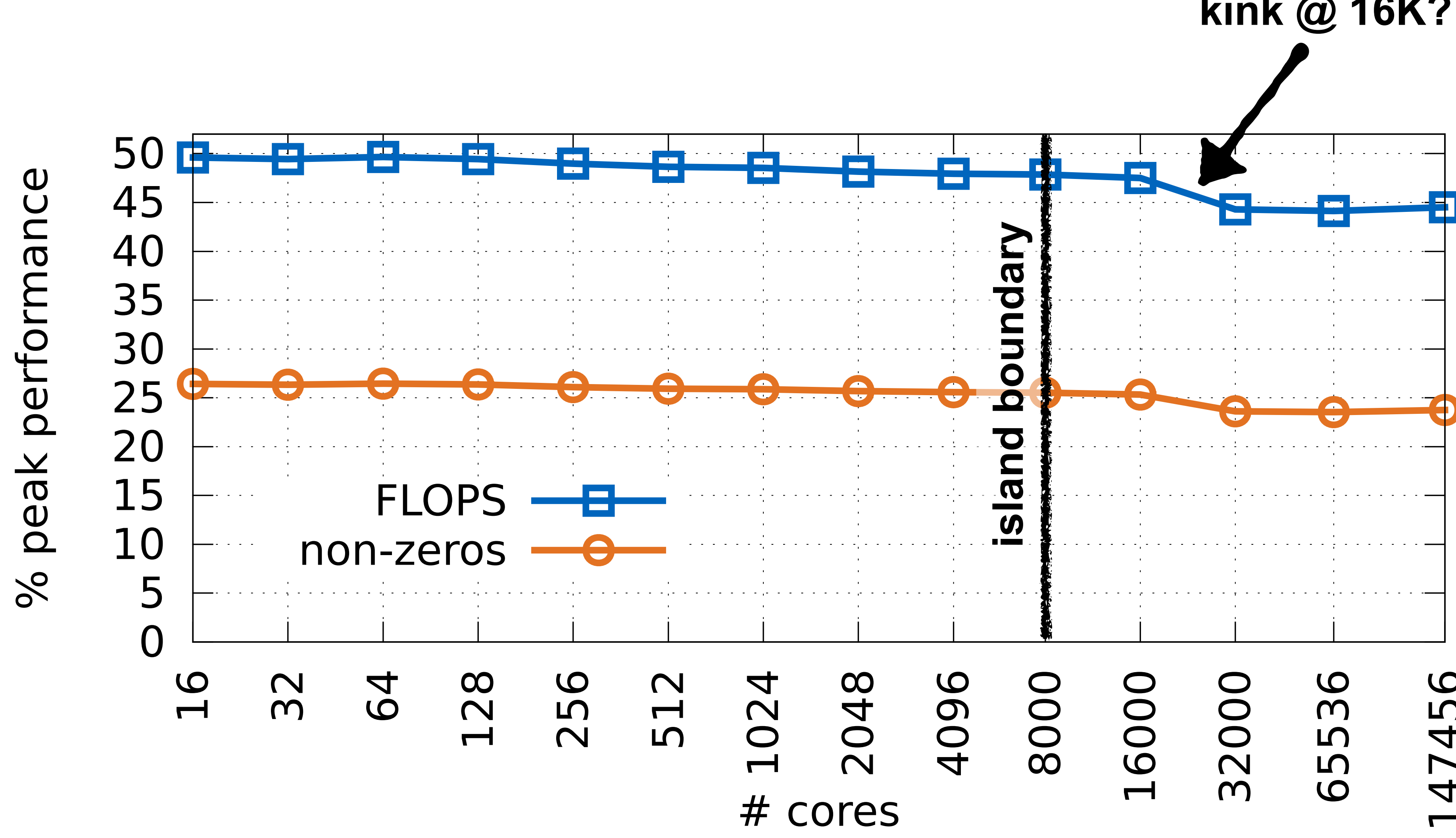
SuperMUC: Weak Scaling

- Cubic domain, 60,000 elements per core
- 100 time steps



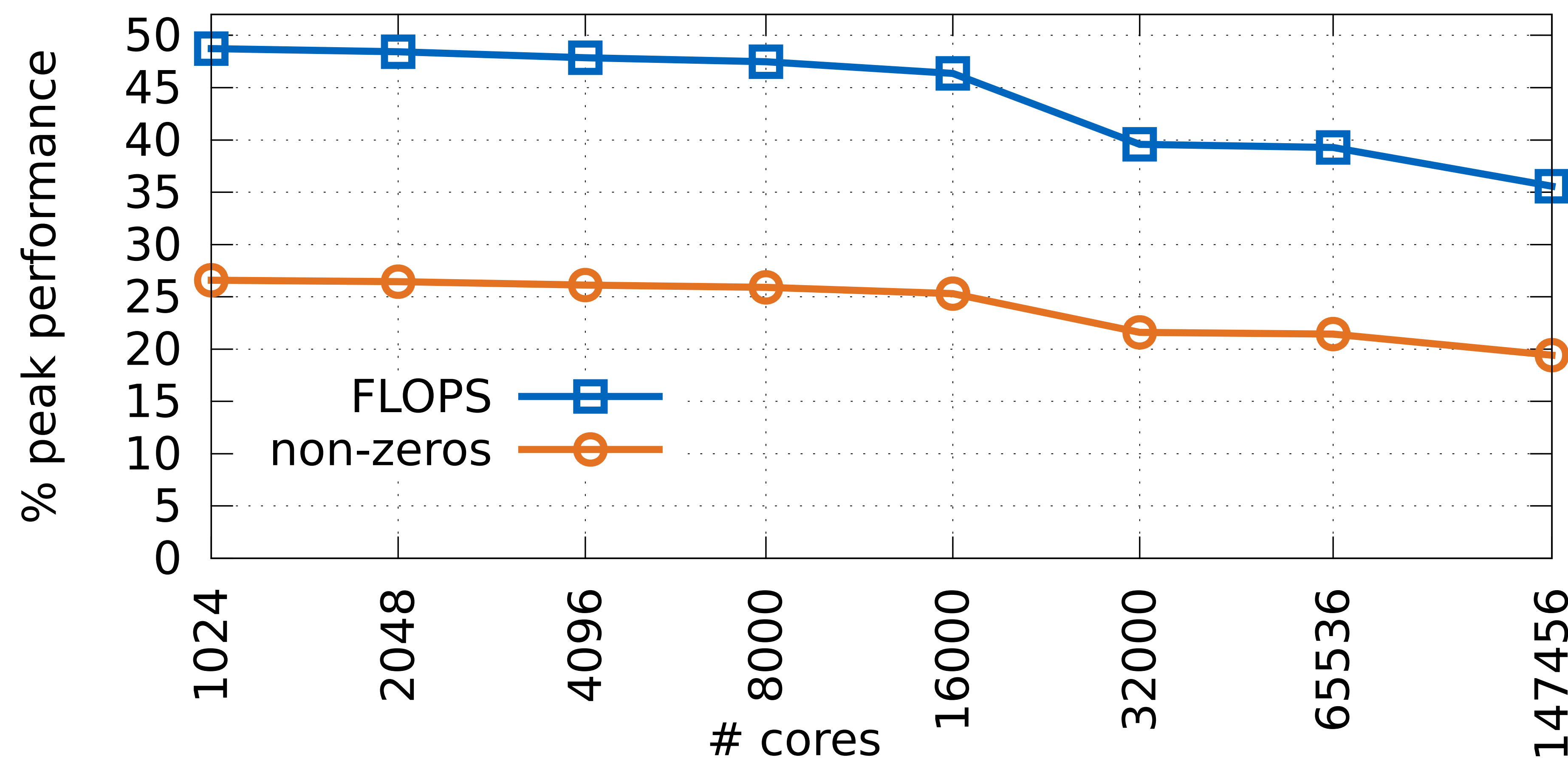
SuperMUC: Weak Scaling

- Cubic domain, 60,000 elements per core
- 100 time steps



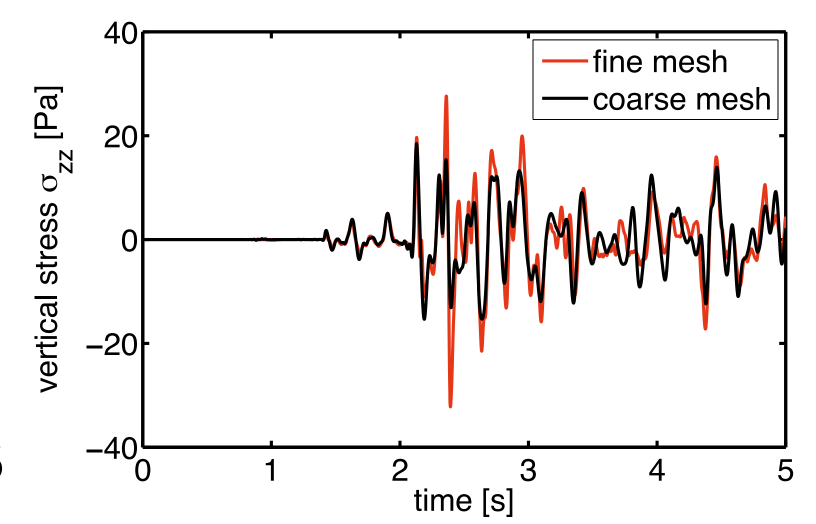
SuperMUC: Strong Scaling

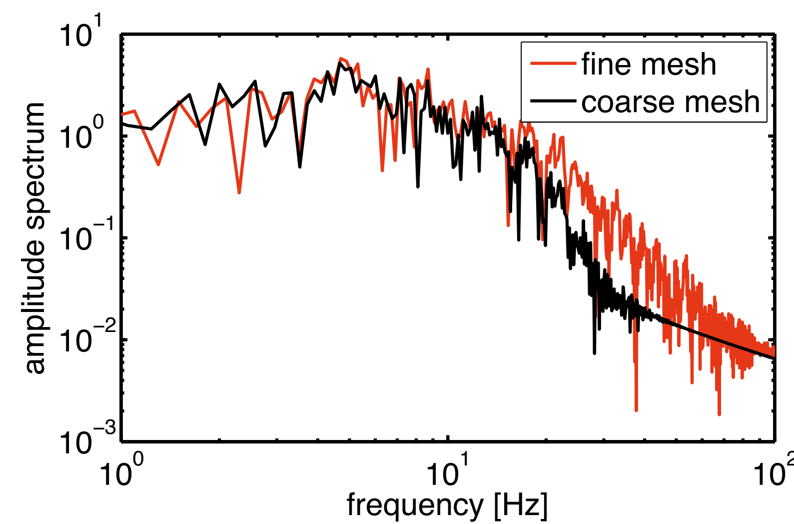
- Mount Merapi; 99,831,401 tetrahedrons (~677 per core for all 147,456 cores)
- 100 time steps

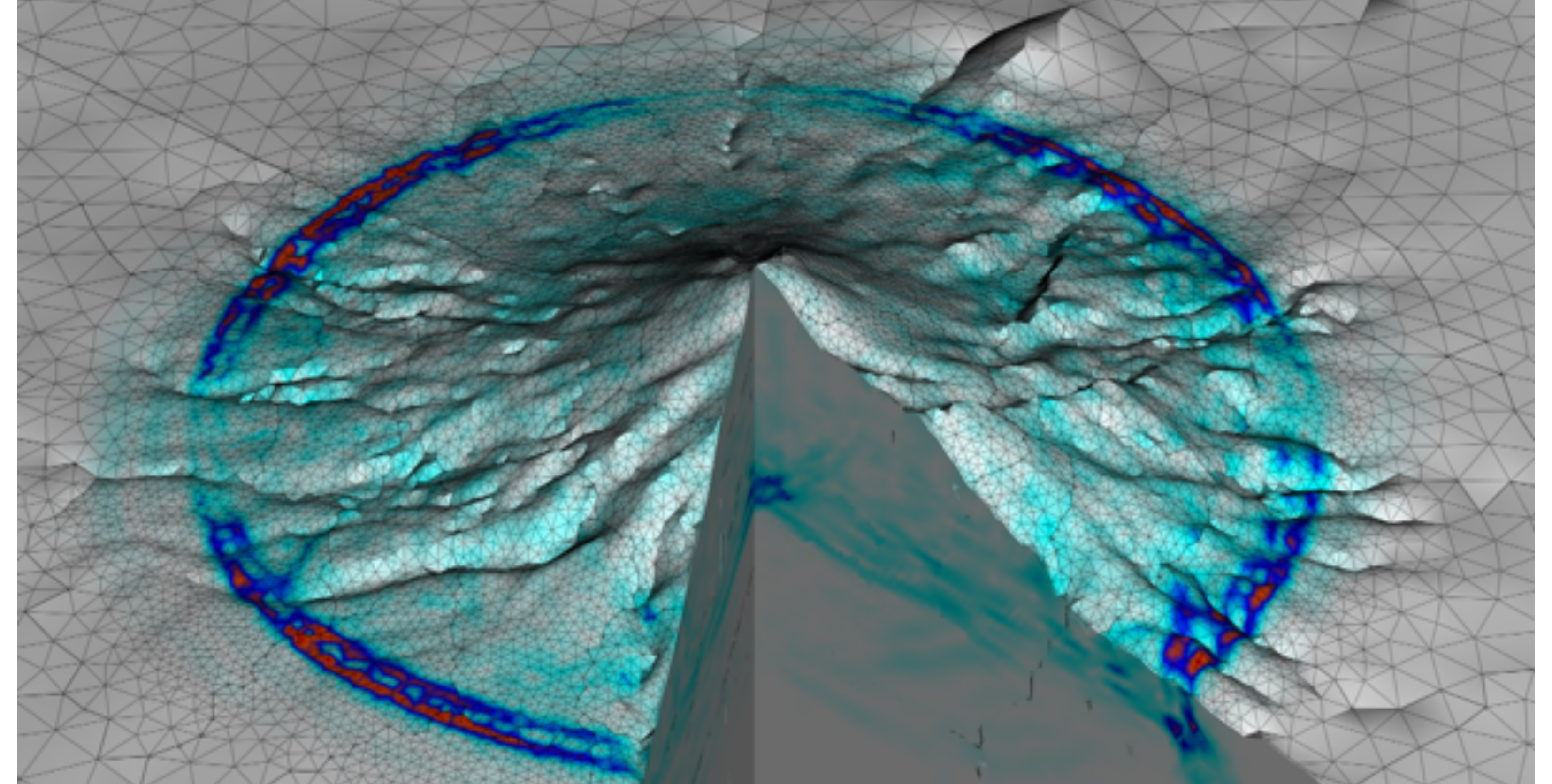


SuperMUC: Production Conditions

- Setting as strong scaling, but:
  - 166,666 time steps: 5 simulation seconds
  - Output: 59 receivers
- 3h 7.5m @ 147,456 SNB-EP cores; initialization 1m 22s
- 1.09 PFLOPS in time stepping loop







Conclusions

- Significant speedup due to kernel optimizations
- Rigorous performance re-engineering process: Sustained petascale application
- SeisSol ready for production runs @ machine level

- Kernels are open-source: https://github.com/TUM-I5/seissol_kernels
- All of SeisSol open-source by this year

Acknowledgements

- Mikhail Smelyanskiy of Intel Labs
- Colleagues of the Leibniz Supercomputing Centre

- Volkswagen Stiftung — Project ASCETE: Advanced Simulation of Coupled Earthquake-Tsunami Events
- Bavarian Competence Network for Technical and Scientific High Performance Computing

- SuperMUC Grant: pr83no
- NSF (Beacon) Grant: 1137097

SuperMUC: Strong Scaling

- SCEC LOH.1, 7252482 elements, 100 time steps
- 6th order in space and time
- 100 time steps