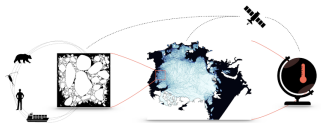
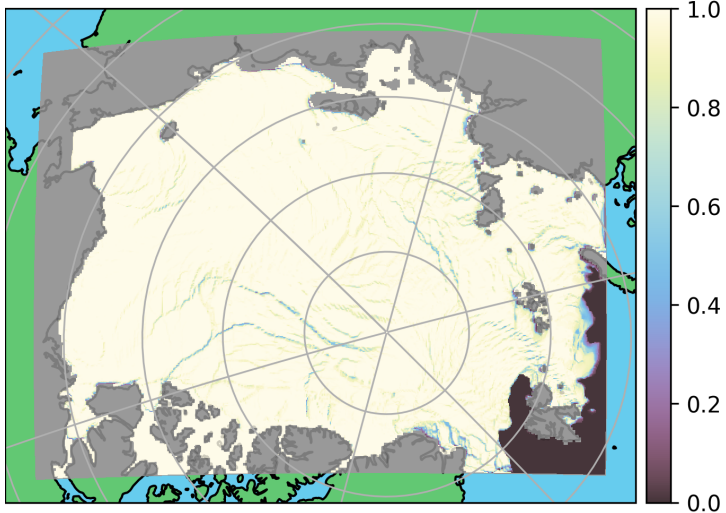


Development of a GPU-accelerated, Finite Element based Dynamical Core for Sea Ice

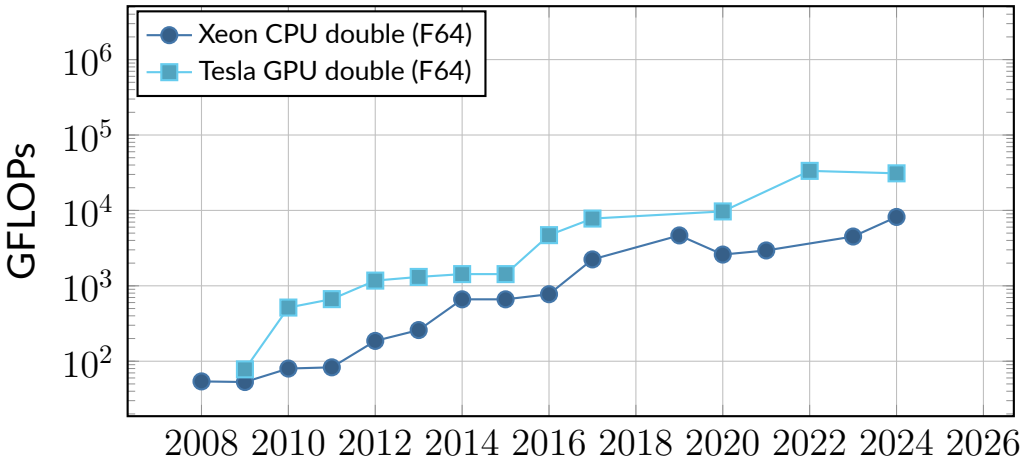
Robert Jendersie [1] Christian Lessig [2] Thomas Richter [1]

[1]Otto-von-Guericke Universität Magdeburg [2]ECMWF Bonn

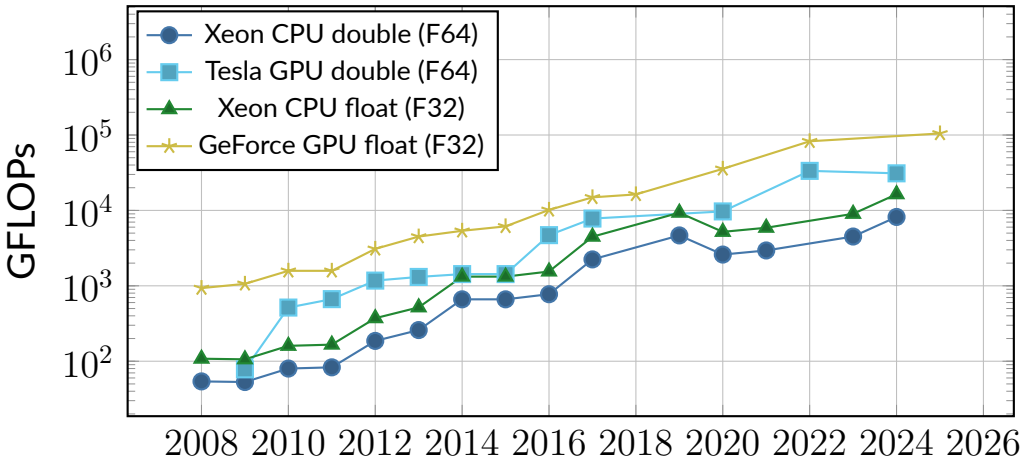
High-Resolution Sea-Ice



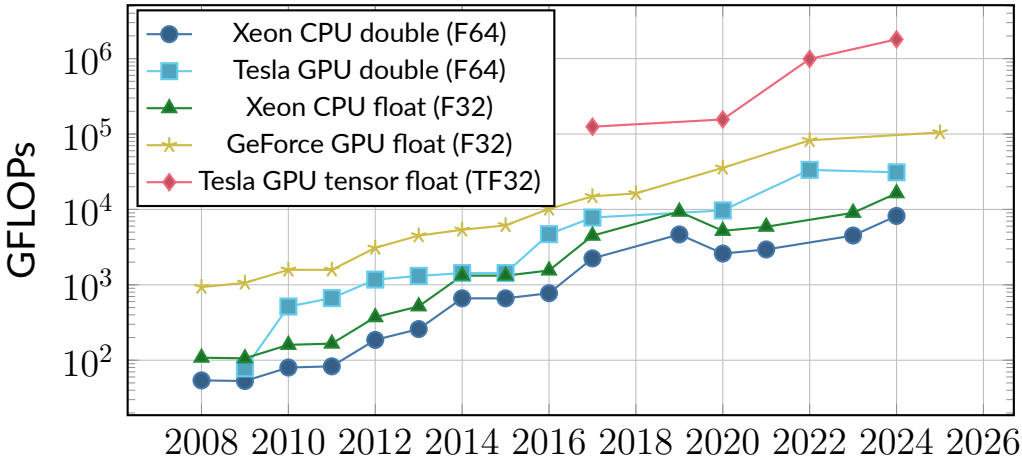
Why GPUs?


Xeon CPU double (F64)
Tesla GPU double (F64)
GFLOPs
10^6
10^5
10^4
10^3
10^2
2008 2010 2012 2014 2016 2018 2020 2022 2024 2026

Why GPUs?


Xeon CPU double (F64)
Tesla GPU double (F64)
Xeon CPU float (F32)
GeForce GPU float (F32)
GFLOPs
10^6
10^5
10^4
10^3
10^2
2008 2010 2012 2014 2016 2018 2020 2022 2024 2026

Why GPUs?


Xeon CPU double (F64)
Tesla GPU double (F64)
Xeon CPU float (F32)
GeForce GPU float (F32)
Tesla GPU tensor float (TF32)
GFLOPs
10^6
10^5
10^4
10^3
10^2
2008 2010 2012 2014 2016 2018 2020 2022 2024 2026

neXtSIM-DG Model

- Momentum and advection

$$\rho_{\text{ice}} H \partial_t \mathbf{v} = \operatorname{div} \boldsymbol{\sigma}(\mathbf{v}, A, H) + F, \tag{1}$$

$$\partial_t H + \operatorname{div}(\mathbf{v}H) = S_H \quad \partial_t A + \operatorname{div}(\mathbf{v}A) = S_A \tag{2}$$

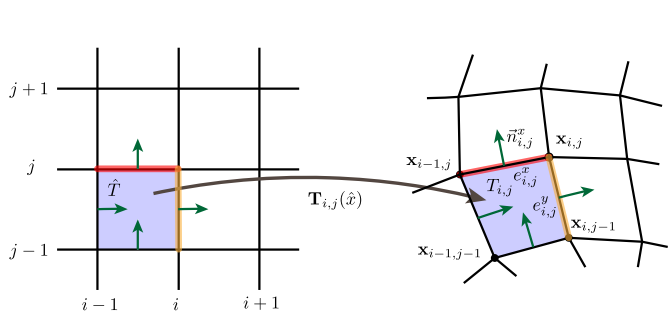
- viscous plastic rheology[1] (with mEVP iteration[2])

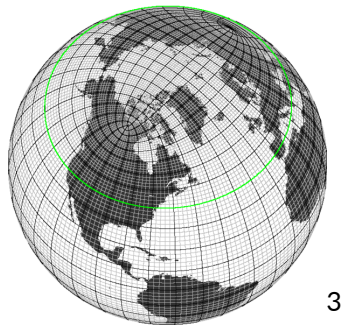
$$\begin{aligned} \boldsymbol{\sigma}^{(p)} &= \frac{\alpha}{1+\alpha}\boldsymbol{\sigma}^{(p-1)} + \frac{1}{1+\alpha}\boldsymbol{\sigma}_{vp}(\mathbf{v}^{(p-1)}, A, H) \\ \boldsymbol{\sigma}_{vp}(\mathbf{v}, A, H) &= \eta(\nabla\mathbf{v} + \nabla\mathbf{v}^T) + \zeta \operatorname{div}(\mathbf{v}) I - \frac{P}{2} I, \end{aligned} \tag{3}$$

[1] Hibler, "A Dynamic Thermodynamic Sea Ice Model", 1979.

[2] Bouillon et al., "The elastic–viscous–plastic method revisited", 2013.

neXtSIM-DG Discretization[4]





[3]

- cG elements for v, dG elements for A, H, σ and explicit time-stepping

[3] `https://www.geomar.de/en/ocean-models`

[4] Richter et al., "A dynamical core based on a discontinuous Galerkin method for higher-order finite-element sea ice modeling", 2023.

Implementation

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // update stress for every element
    for (int i = 0; i < N; ++i)    {
        Vector h      = max{0, H_{i,*} PSI_A}
        Vector a      = min{1, max{0, A_{i,*} PSI_A}}
        Vector e^11 = E^11_{i,*} PSI_S
        Vector e^12 = E^12_{i,*} PSI_S
        Vector e^22 = E^22_{i,*} PSI_S

        Vector P_D = P* · h * exp(-20(1-a))
            *(Δ^2_min + 5/4(E^11_{i,*} * E^11_{i,*} + E^22_{i,*} * E^22_{i,*}) + 3/2 E^11_{i,*} * E^22_{i,*} + E^12_{i,*} * E^12_{i,*})^{-1/2}

        S^11_{i,*} = (1-α^{-1}) S^11_{i,*} + α^{-1} T_i^{-1} (P_D * (5/8 e^11 + 3/8 e^22) - 1/2 P)
        S^12_{i,*} = (1-α^{-1}) S^12_{i,*} + α^{-1} T_i^{-1} (P_D * 1/4 e^12)
        S^22_{i,*} = (1-α^{-1}) S^22_{i,*} + α^{-1} T_i^{-1} (P_D * (5/8 e^22 + 3/8 e^11) - 1/2 P)
    }
}
```

Implementation

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,   // input fields N × DoF
    const Matrix& H, const Matrix& A) {
    // update stress for every element
    for (int i = 0; i < N; ++i)  {
        Vector h      = max{0, H_{i,*} PSI_A}
        Vector a      = min{1, max{0, A_{i,*} PSI_A}}
        Vector e^11 = E^11_{i,*} PSI_S
        Vector e^12 = E^12_{i,*} PSI_S
        Vector e^22 = E^22_{i,*} PSI_S

        Vector P_D = P* · h * exp(-20(1-a))
            *(Δ^2_min + 5/4(E^11_{i,*} * E^11_{i,*} + E^22_{i,*} * E^22_{i,*}) + 3/2 E^11_{i,*} * E^22_{i,*} + E^12_{i,*} * E^12_{i,*})^{-1/2}

        S^11_{i,*} = (1-α^{-1}) S^11_{i,*} + α^{-1} T_i^{-1} (P_D * (5/8 e^11 + 3/8 e^22) - 1/2 P)
        S^12_{i,*} = (1-α^{-1}) S^12_{i,*} + α^{-1} T_i^{-1} (P_D * 1/4 e^12)
        S^22_{i,*} = (1-α^{-1}) S^22_{i,*} + α^{-1} T_i^{-1} (P_D * (5/8 e^22 + 3/8 e^11) - 1/2 P)
    }
}
```

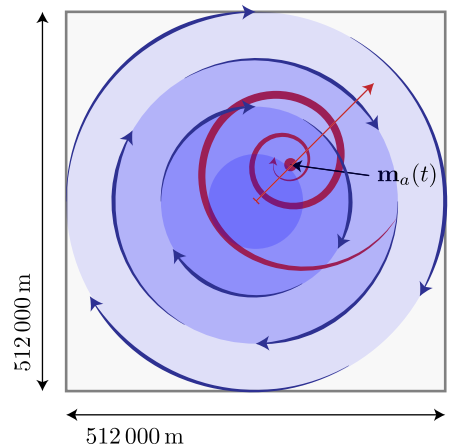
Implementation

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // update stress for every element
    for (int i = 0; i < N; ++i)  {
        Vector h     = max{0, H_{i,*} PSI_A}
        Vector a     = min{1, max{0, A_{i,*} PSI_A}}
        Vector e^11  = E^11_{i,*} PSI_S                    // eval fields at gauss points
        Vector e^12  = E^12_{i,*} PSI_S
        Vector e^22  = E^22_{i,*} PSI_S

        Vector P_D   = P* · h * exp(−20(1 − a))
            *(Δ²_min + 5/4(E^11_{i,*} * E^11_{i,*} + E^22_{i,*} * E^22_{i,*}) + 3/2 E^11_{i,*} * E^22_{i,*} + E^12_{i,*} * E^12_{i,*})^{−1/2}

        S^11_{i,*} = (1 − α^{−1}) S^11_{i,*} + α^{−1} T_i^{−1} (P_D * (5/8 e^11 + 3/8 e^22) − 1/2 P)
        S^12_{i,*} = (1 − α^{−1}) S^12_{i,*} + α^{−1} T_i^{−1} (P_D * 1/4 e^12)
        S^22_{i,*} = (1 − α^{−1}) S^22_{i,*} + α^{−1} T_i^{−1} (P_D * (5/8 e^22 + 3/8 e^11) − 1/2 P)
    }
}
```

Implementation

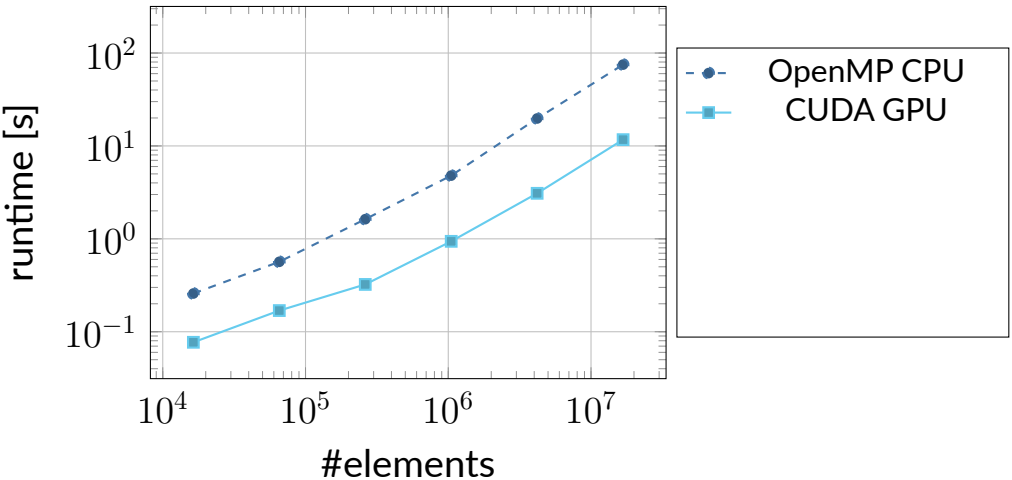
```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // update stress for every element
    for (int i = 0; i < N; ++i)  {
        Vector h    = max{0, H_{i,*} PSI_A}
        Vector a    = min{1, max{0, A_{i,*} PSI_A}}
        Vector e^11 = E^11_{i,*} PSI_S                    // eval fields at gauss points
        Vector e^12 = E^12_{i,*} PSI_S
        Vector e^22 = E^22_{i,*} PSI_S

        Vector P_D  = P* · h * exp(-20(1-a))
            *(Δ^2_min + 5/4(E^11_{i,*} * E^11_{i,*} + E^22_{i,*} * E^22_{i,*}) + 3/2 E^11_{i,*} * E^22_{i,*} + E^12_{i,*} * E^12_{i,*})^{-1/2}

        S^11_{i,*} = (1-α^{-1}) S^11_{i,*} + α^{-1} T_i^{-1} (P_D * (5/8 e^11 + 3/8 e^22) - 1/2 P)
        S^12_{i,*} = (1-α^{-1}) S^12_{i,*} + α^{-1} T_i^{-1} (P_D * 1/4 e^12)          // mapping of parametric mesh
        S^22_{i,*} = (1-α^{-1}) S^22_{i,*} + α^{-1} T_i^{-1} (P_D * (5/8 e^22 + 3/8 e^11) - 1/2 P)
    }
}
```

Benchmark - Cyclone in a Box[5]

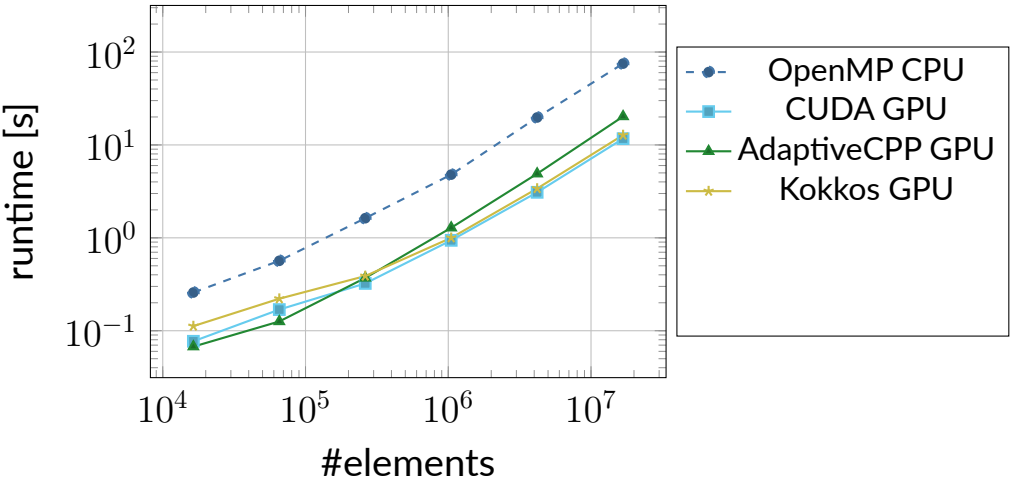


[5] Mehlmann et al., “Simulating Linear Kinematic Features in Viscous-Plastic Sea Ice Models on Quadrilateral and Triangular Grids With Different Variable Staggering”, 2021.

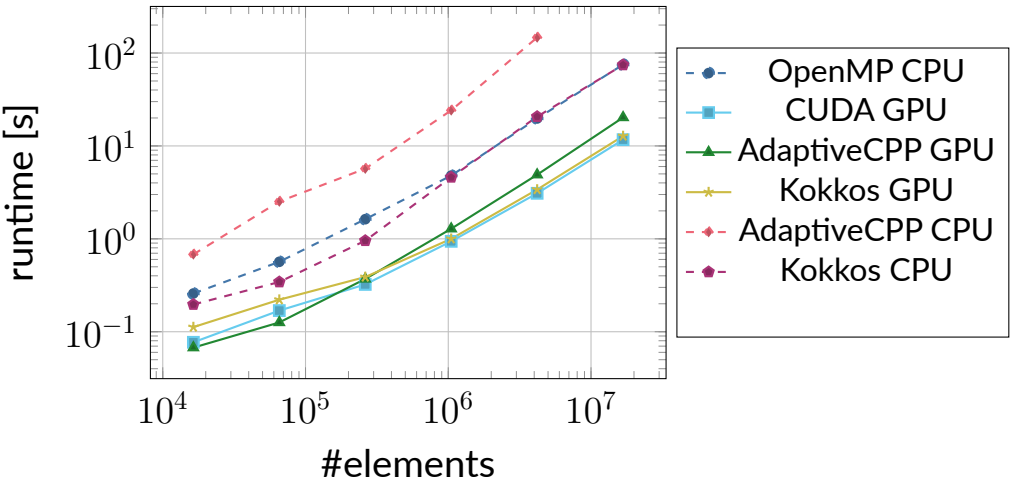
Comparison F64 - A100 / $2\times$ EPYC 7402, 48 cores



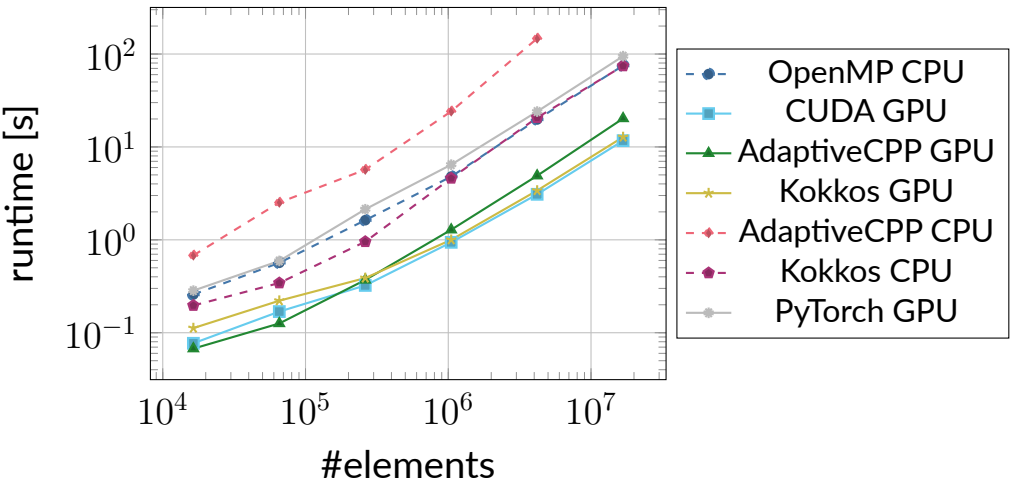
Comparison F64 - A100 / $2\times$ EPYC 7402, 48 cores



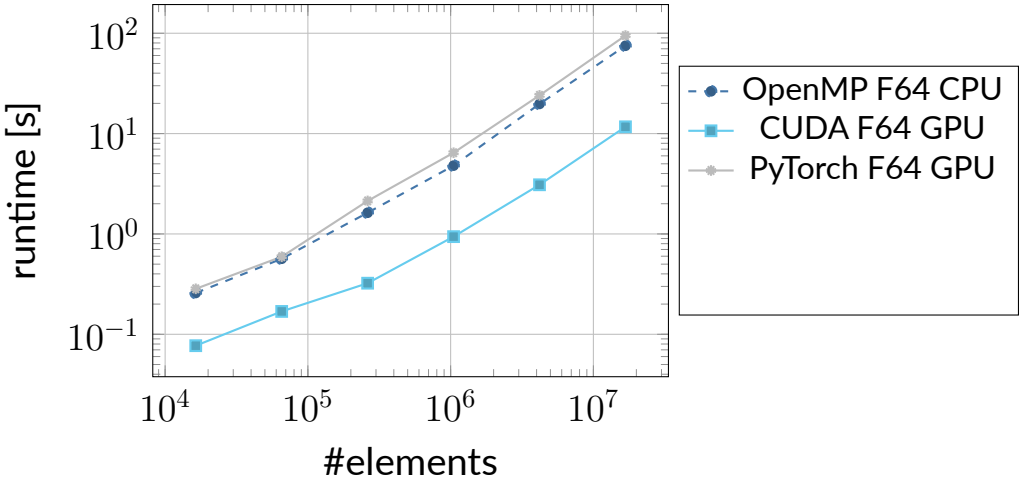
Comparison F64 - A100 / $2\times$ EPYC 7402, 48 cores



Comparison F64 - A100 / $2\times$ EPYC 7402, 48 cores

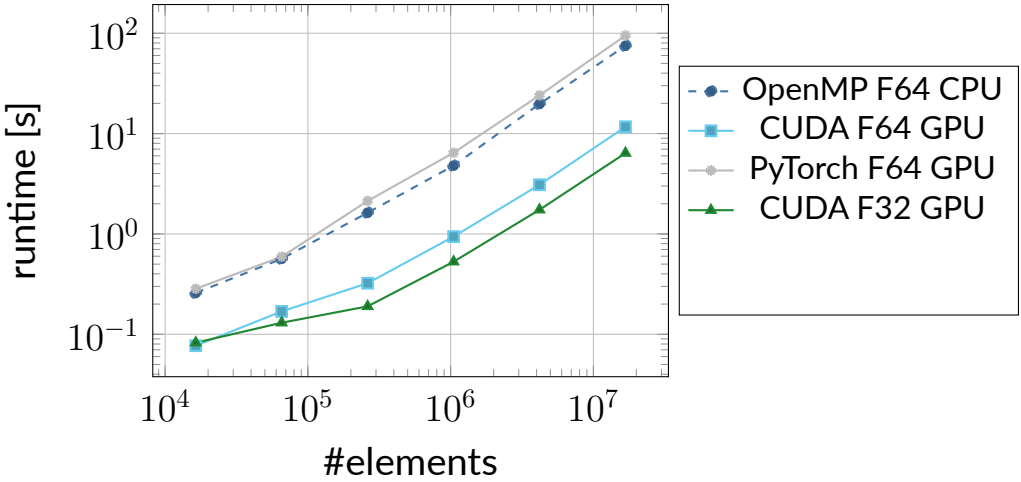


Comparison - Mixed Precision[6]



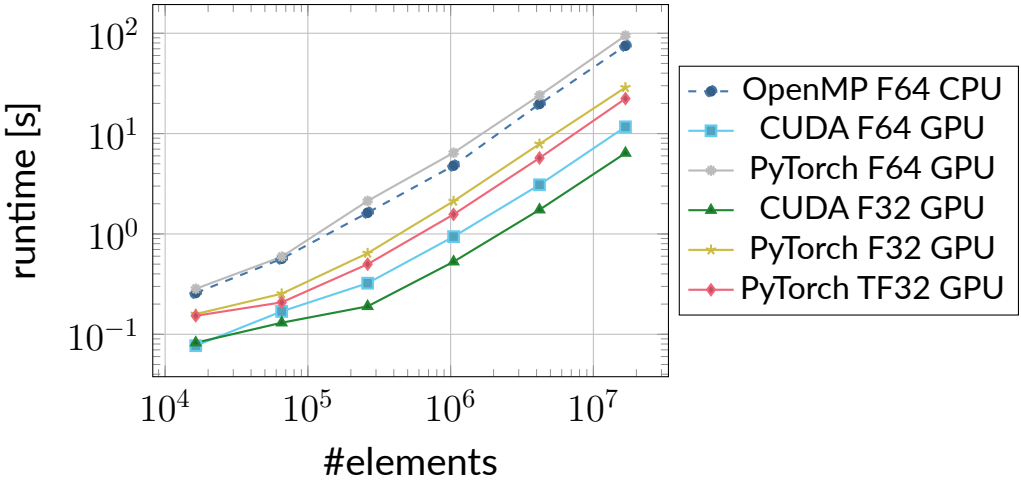
[6]Accuracy investigated in: Jendersie, Lessig, and Richter, "A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)", 2024

Comparison - Mixed Precision[6]



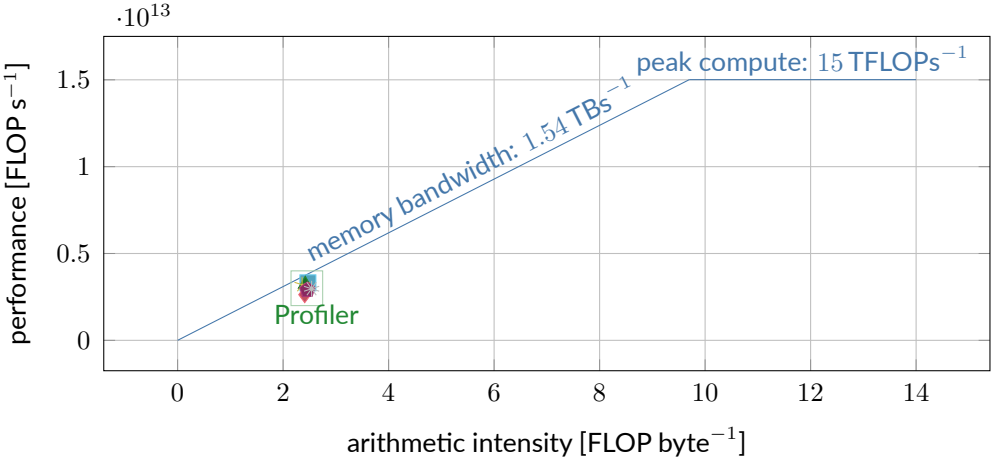
[6]Accuracy investigated in: Jendersie, Lessig, and Richter, "A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)", 2024

Comparison - Mixed Precision[6]

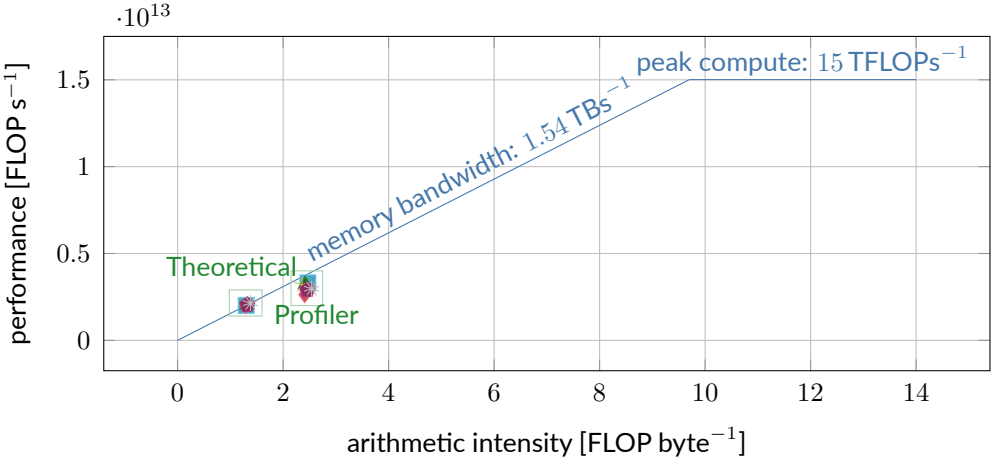


[6]Accuracy investigated in: Jendersie, Lessig, and Richter, "A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)", 2024

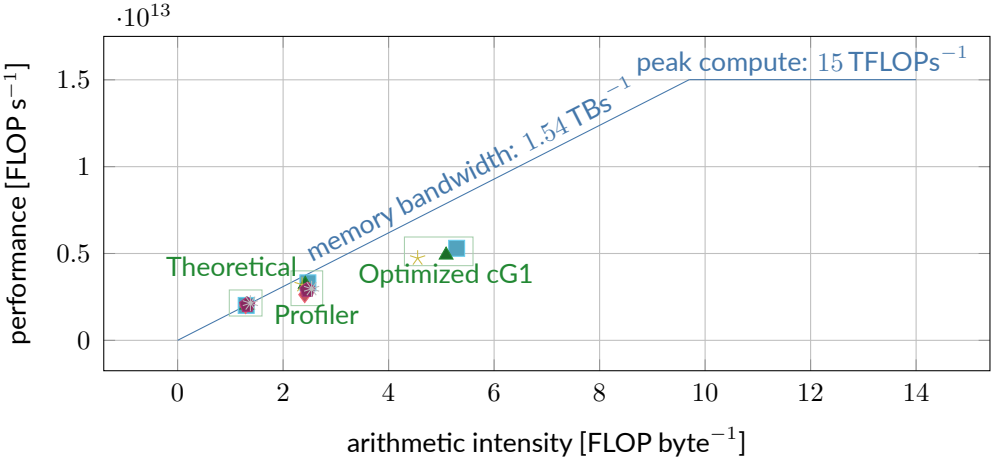
Roofline Analysis - A100



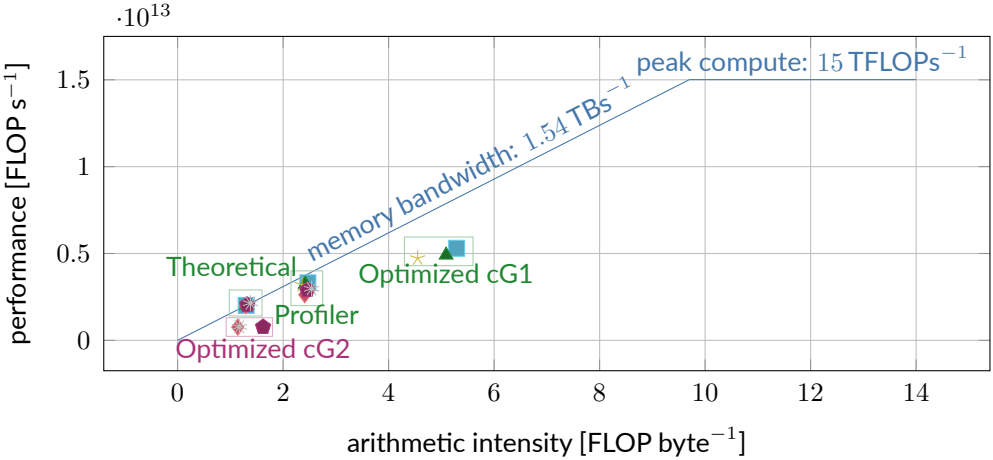
Roofline Analysis - A100



Roofline Analysis - A100



Roofline Analysis - A100



Choice for Full Implementation

Performance

$$\text{CUDA} \leq \textbf{Kokkos} < \text{AdaptiveCPP(Sycl)} < \text{PyTorch}$$

Choice for Full Implementation

Performance

$$\text{CUDA} \leq \textbf{Kokkos} < \text{AdaptiveCPP(Sycl)} < \text{PyTorch}$$

Ease of Development

$$\text{PyTorch} < \text{Sycl} < \textbf{Kokkos} < \text{CUDA}$$

Choice for Full Implementation

Performance

$$\text{CUDA} \leq \textbf{Kokkos} < \text{AdaptiveCPP(Sycl)} < \text{PyTorch}$$

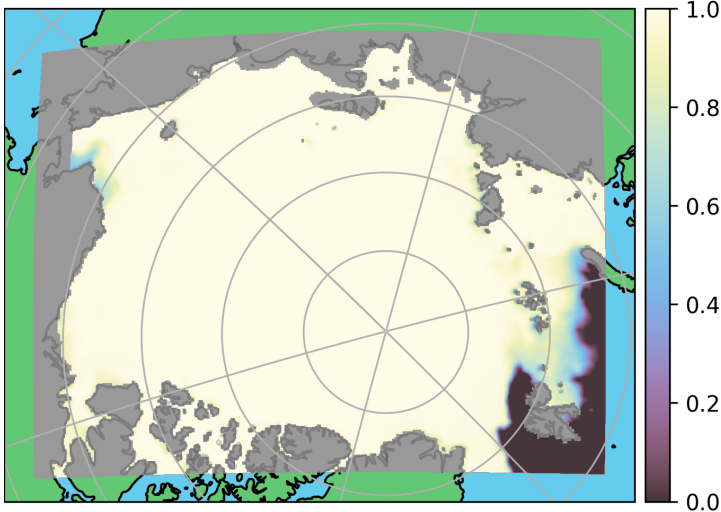
Ease of Development

$$\text{PyTorch} < \text{Sycl} < \textbf{Kokkos} < \text{CUDA}$$

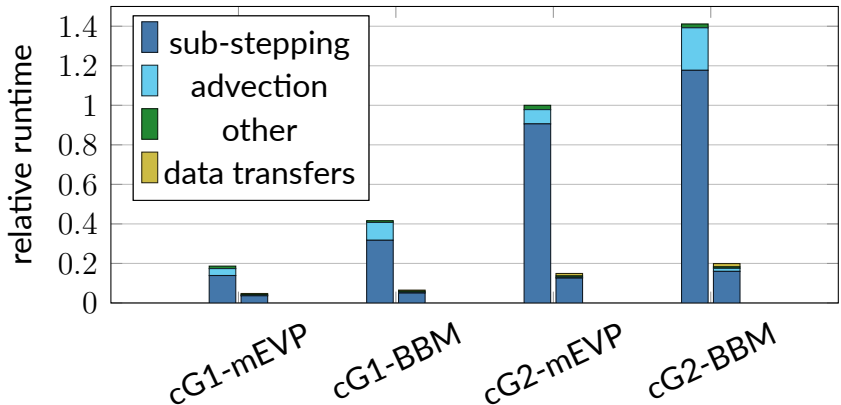
Deployment effort

$$\text{CUDA} \leq \textbf{Kokkos} < \text{PyTorch} < \text{AdaptiveCPP(Sycl)}$$

Test case - Arctic



Performance - Arctic 3.125 km



2 x EPYC 7773X, 122 cores (left) - H100 PCIe GPU (right)

Conclusion

- Kokkos can be decent alternative to CUDA
- Higher order CG/DG discretization works well on GPU
- Effective use of tensor cores is difficult

Robert Jendersie, Christian Lessig, and Thomas Richter. “A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)”. In: (Sept. 2024). DOI: 10.5194/egusphere-2024-2539

This project is supported by Schmidt Sciences, LLC.

Conclusion

- Kokkos can be decent alternative to CUDA
- Higher order CG/DG discretization works well on GPU
- Effective use of tensor cores is difficult

Robert Jendersie, Christian Lessig, and Thomas Richter. "A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)". In: (Sept. 2024). DOI: 10.5194/egusphere-2024-2539

This project is supported by Schmidt Sciences, LLC.

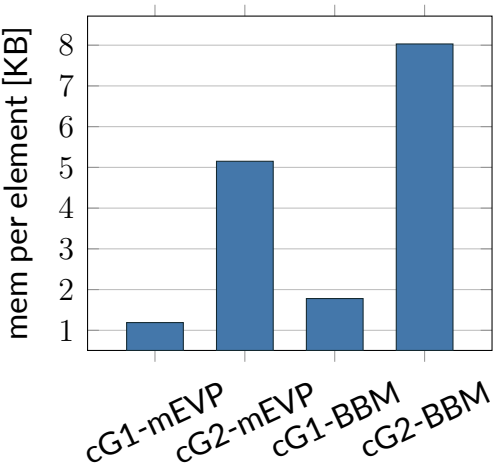
Conclusion

- Kokkos can be decent alternative to CUDA
- Higher order CG/DG discretization works well on GPU
- Effective use of tensor cores is difficult

Robert Jendersie, Christian Lessig, and Thomas Richter. "A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)". In: (Sept. 2024). DOI: 10.5194/egusphere-2024-2539

This project is supported by Schmidt Sciences, LLC.

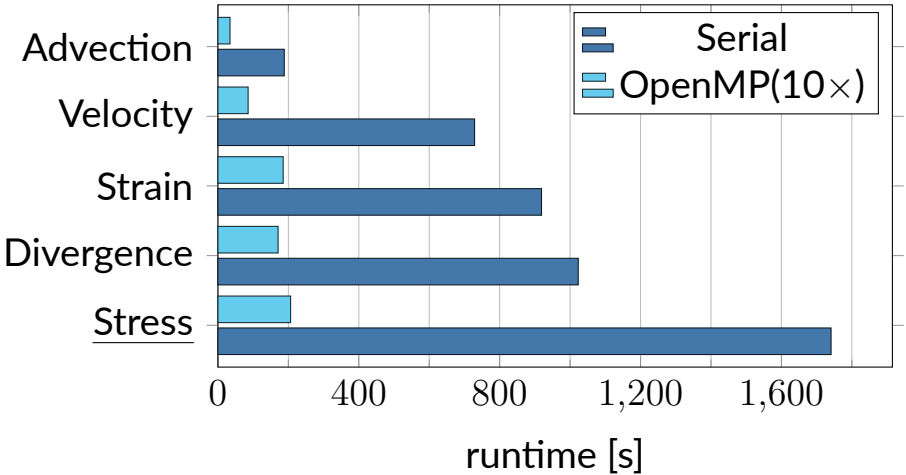
Device Memory Usage



max #elem with 40GB VRAM:

- cG1-mEVP 3.3×10^7
- cG2-BBM 4.9×10^6

Parallelization



Frameworks

- Directive based: OpenMP, OpenACC
- Low level interface: CUDA
- Heterogeneous compute: Sycl, Kokkos
- Machine learning framework: PyTorch

Implementation OpenMP

```
void StressUpdateHighOrder(Matrix& S11, Matrix& S12, Matrix& S22,
    const Matrix& E11, const Matrix& E12, const Matrix& E22,
    const Matrix& H, const Matrix& A) {
    // update stress in parallel
#pragma omp parallel for
    for (int i=0; i<N; ++i) {
        computeStress(i, S11, S12, S22, E11, E12,E22, H, A, α);
    }
}
```

Implementation OpenMP

```
void StressUpdateHighOrder(Matrix& S11, Matrix& S12, Matrix& S22,
    const Matrix& E11, const Matrix& E12, const Matrix& E22,
    const Matrix& H, const Matrix& A) {
    // update stress in parallel
#pragma omp target parallel for
    for (int i=0; i<N; ++i) {
        computeStress(i, S11, S12, S22, E11, E12,E22, H, A, α);
    }
}
```

Implementation CUDA

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        double* Ŝ^11 = cudaAlloc(N · n_s · sizeof(double))
        ...
    }
    // upload data
    cudaCopy(S^11, Ŝ^11, N · n_s · sizeof(double))
    ...
    // invoke kernel
    computeStress<<<N_grid, N_block>>>(Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α);
    // fetch data
    cudaCopy(Ŝ^11, S^11, N · n_s · sizeof(double))
    ...
}
```

Implementation CUDA

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        double* Ŝ^11 = cudaAlloc(N · n_s · sizeof(double))
        ...
    }
    // upload data
    cudaCopy(S^11, Ŝ^11, N · n_s · sizeof(double))
    ...
    // invoke kernel
    computeStress<<<N_grid, N_block>>>(Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α);
    // fetch data
    cudaCopy(Ŝ^11, S^11, N · n_s · sizeof(double))
    ...
}
```

Implementation CUDA

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        double* Ŝ^11 = cudaAlloc(N · n_s · sizeof(double))
        ...
    }
    // upload data
    cudaCopy(S^11, Ŝ^11, N · n_s · sizeof(double))
    ...
    // invoke kernel
    computeStress<<<N_grid, N_block>>>(Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α);
    // fetch data
    cudaCopy(Ŝ^11, S^11, N · n_s · sizeof(double))
    ...
}
```

Implementation Kokkos

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        auto Ŝ^11 = View(N,N_s)
        ...
    }
    // upload data
    kokkosCopy(S^11, Ŝ^11)
    ...
    // compute in parallel
    parallel_for(N, KOKKOS_CLASS_LAMBDA(int i){
        stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12,Ê^22, Ĥ, Â, α)
    })
    // fetch data
    kokkosCopy(Ŝ^11, S^11)
    ...
}
```

Implementation Kokkos

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        auto Ŝ^11 = View(N,N_s)
        ...
    }
    // upload data
    kokkosCopy(S^11, Ŝ^11)
    ...
    // compute in parallel
    parallel_for(N, KOKKOS_CLASS_LAMBDA(int i){
        stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12,Ê^22, Ĥ, Â, α)
    })
    // fetch data
    kokkosCopy(Ŝ^11, S^11)
    ...
}
```

Implementation Kokkos

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        auto Ŝ^11 = View(N, N_s)
        ...
    }
    // upload data
    kokkosCopy(S^11, Ŝ^11)
    ...
    // compute in parallel
    parallel_for(N, KOKKOS_CLASS_LAMBDA(int i){
        stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α)
    })
    // fetch data
    kokkosCopy(Ŝ^11, S^11)
    ...
}
```

Implementation Sycl

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // define buffers
    if (init) {
        auto S̄^11 = buffer(S^11, N, N_s)
        ...
    }
    queue.submit([&]() {
        // declare required data
        auto Ŝ^11 = accessor(S̄^11, read_write)
        ...
        // compute in parallel
        parallel_for(N, [=](int i) {
            stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α)
        })
    }
}
```

Implementation Sycl

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // define buffers
    if (init) {
        auto S̄^11 = buffer(S^11, N, N_s)
        ...
    }
    queue.submit([&]() {
        // declare required data
        auto Ŝ^11 = accessor(S̄^11, read_write)
        ...
        // compute in parallel
        parallel_for(N, [=](int i) {
            stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α)
        })
    }
}
```

Implementation Sycl

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // define buffers
    if (init) {
        auto S̄^11 = buffer(S^11, N, N_s)
        ...
    }
    queue.submit([&]() {
        // declare required data
        auto Ŝ^11 = accessor(S̄^11, read_write)
        ...
        // compute in parallel
        parallel_for(N, [=](int i) {
            stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α)
        })
    }
}
```

Implementation Sycl

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // define buffers
    if (init) {
        auto S̄^11 = buffer(S^11, N, N_s)
        ...
    }
    queue.submit([&]() {
        // declare required data
        auto Ŝ^11 = accessor(S̄^11, read_write)
        ...
        // compute in parallel
        parallel_for(N, [=](int i) {
            stressUpdate(i, Ŝ^11, Ŝ^12, Ŝ^22, Ê^11, Ê^12, Ê^22, Ĥ, Â, α)
        })
    }
}
```

Implementation PyTorch

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        auto Ŝ^11 = Tensor(N, N_s, "cuda")
        ...
    }
    // upload data
    Ŝ^11.copy_(S^11)
    ...
    // compute on device
    Tensor h = max{0, ĤPSI}
    Tensor a = min{1, max{0, ÂPSI}}
    ...
    // fetch data
    S^11.copy_(Ŝ^11)
    ...
}
```

Implementation PyTorch

```
void StressUpdateHighOrder(Matrix& S^11, Matrix& S^12, Matrix& S^22,
    const Matrix& E^11, const Matrix& E^12, const Matrix& E^22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        auto Ŝ^11 = Tensor(N, N_s, "cuda")
        ...
    }
    // upload data
    Ŝ^11.copy_(S^11)
    ...
    // compute on device
    Tensor h = max{0, ĤPSI}
    Tensor a = min{1, max{0, ÂPSI}}
    ...
    // fetch data
    S^11.copy_(Ŝ^11)
    ...
}
```

Implementation PyTorch

```
void StressUpdateHighOrder(Matrix& S11, Matrix& S12, Matrix& S22,
    const Matrix& E11, const Matrix& E12, const Matrix& E22,
    const Matrix& H, const Matrix& A) {
    // allocate memory
    if (init) {
        auto Ŝ11 = Tensor(N, Ns, "cuda")
        ...
    }
    // upload data
    Ŝ11.copy_(S11)
    ...
    // compute on device
    Tensor h = max{0, ĤPSI}
    Tensor a = min{1, max{0, ÂPSI}}
    ...
    // fetch data
    S11.copy_(Ŝ11)
    ...
}
```

Development and Deployment

Ease of Development

$$\text{PyTorch} < \text{Sycl} < \text{Kokkos} < \text{CUDA}$$

Deployment effort

$$\text{CUDA} \leq \text{Kokkos} < \text{PyTorch} < \text{Sycl(AdaptiveCPP)}$$

Development and Deployment

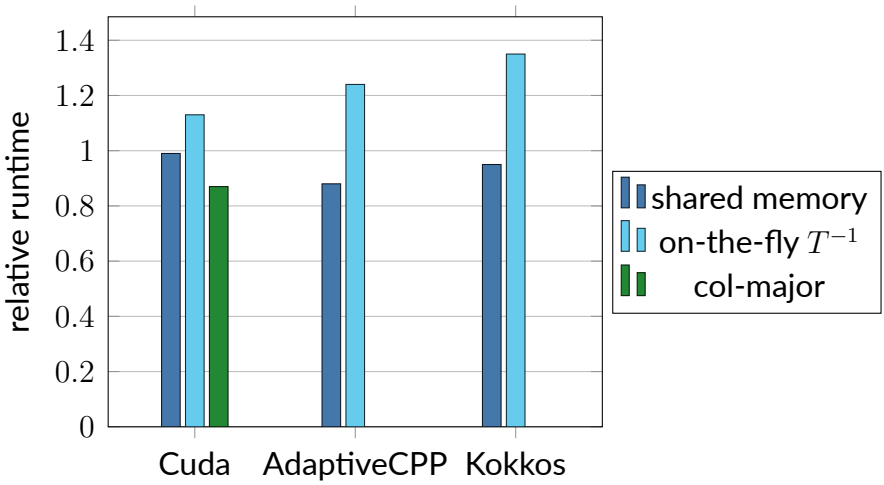
Ease of Development

$$\text{PyTorch} < \text{Sycl} < \text{Kokkos} < \text{CUDA}$$

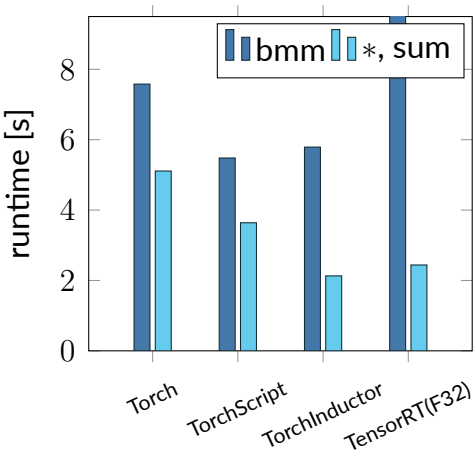
Deployment effort

$$\text{CUDA} \leq \text{Kokkos} < \text{PyTorch} < \text{Sycl(AdaptiveCPP)}$$

Optimization

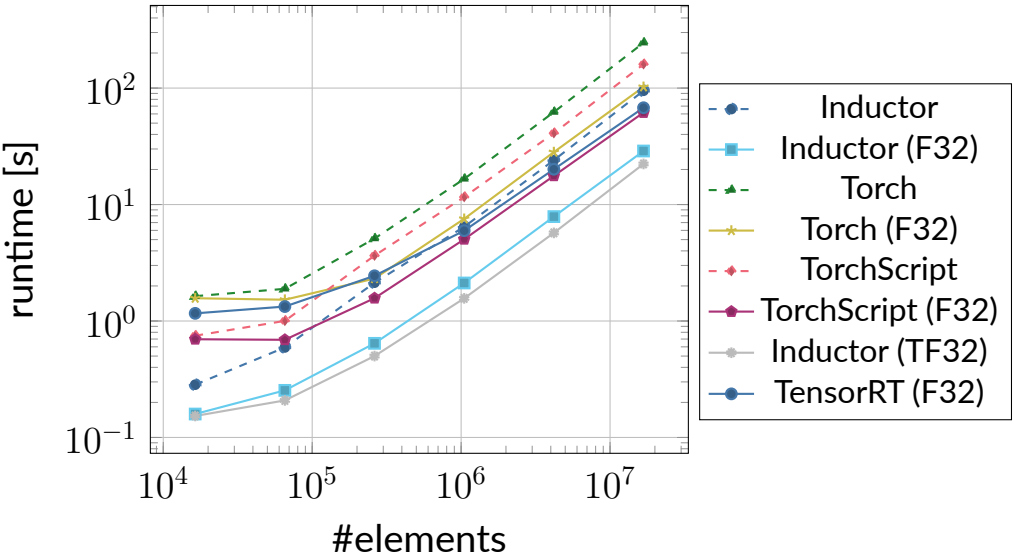


Optimization - PyTorch




```
# m1 ∈ ℝ^{262144×4×3} , m2 ∈ ℝ^{262144×3}
# bmm
def bmm(m1, m2):
    return torch.bmm(m1,m2.unsqueeze_(2))
        .squeeze_(2)
# *,sum
def mulsum(m1, m2):
    return torch.sum(m1 * m2.unsqueeze_(1),
        dim=2)
```

Performance - PyTorch



Performance - Higher Order