Performance Optimization of Matrix-free Finite-Element Algorithms within deal.II

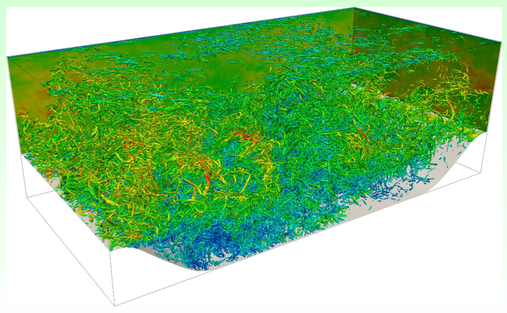
M. Kronbichler[a], K. Ljungkvist[b], M. Allalen[c], M. Ohlerich[c], I. Pasychnik[d], W. A. Wall[a]

[a] Institute for Computational Mechanics, Technical University of Munich, Germany, [b] Uppsala University, Sweden, [c] Leibniz Computing Centre, Garching, Germany, [d] IBM Germany

Summary

- Matrix-free finite element kernels in `deal.II` library (`www.dealii.org`) ported to Xeon Phi and P100, support for geometric multigrid on adaptively refined meshes
- New developments for `deal.II` library: tuned AVX-512 vectorization for continuous elements, GPU code with matrix-free functionality
- Analysis of highly tuned matrix-vector product as proxy for application performance in fluid dynamics [1]
- P100 $2\times$ faster than KNL, $4\times$ faster than Haswell & Broadwell if memory bound; similar on Cartesian meshes (up to 300 GFLOP/s)
- NVLink communication on multiple GPUs with MPI-like setup: explicitly send ghost data, overlapped with computations

Application background: Simulation of turbulent flow past periodic hills with 450 million degrees of freedom, flow field visualized by Q-criterion



Matrix-free Algorithm Description

Matrix-free algorithm in finite element programs exchanges **matrix-vector product** in matrix-based scheme

$$\begin{cases} A = \sum_{K\in\{\text{cells}\}} P_K^T A_K P_K \text{ (with assembly)} \\ v = Au \text{ (sparse mat-vec within iterative solver)} \end{cases}$$

by evaluation of integrals **within the iterative solver**:

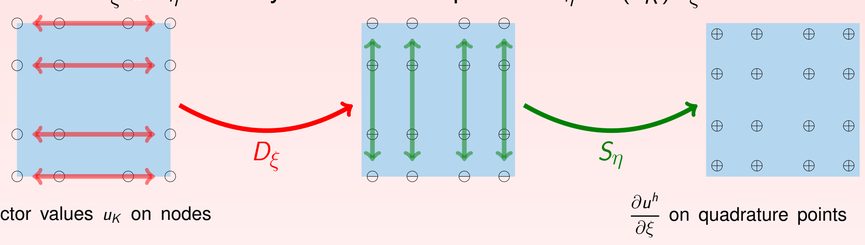
$$v = \sum_{K\in\{\text{cells}\}} P_K^T A_K (P_K u)$$

Matrix-free algorithm:

- $v = 0$
- loop over cells
  - (i) Extract local vector values on cell: $u_K = P_K u$
  - (ii) Apply operation locally on cell: $v_K = A_K u_K$ (**without** forming A_K)
  - (iii) Sum results from (ii) into the global solution vector: $v = v + P_K^T v_K$

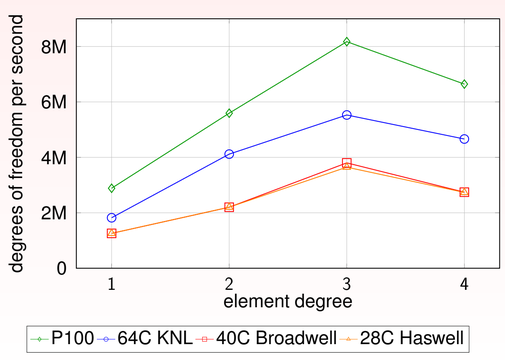
Local evaluation of integrals: Sum factorization on quadrilaterals/hexahedra through `deal.II` finite element library `www.dealii.org` [2, 3, 4].

Example for evaluation of $\frac{\partial u}{\partial \xi}$ in all quadrature points, given node values u_K with interpolation matrix $D_\xi \otimes S_\eta$ done by matrix-matrix product $S_\eta \text{mat}(u_K) D_\xi$:



Multigrid application performance

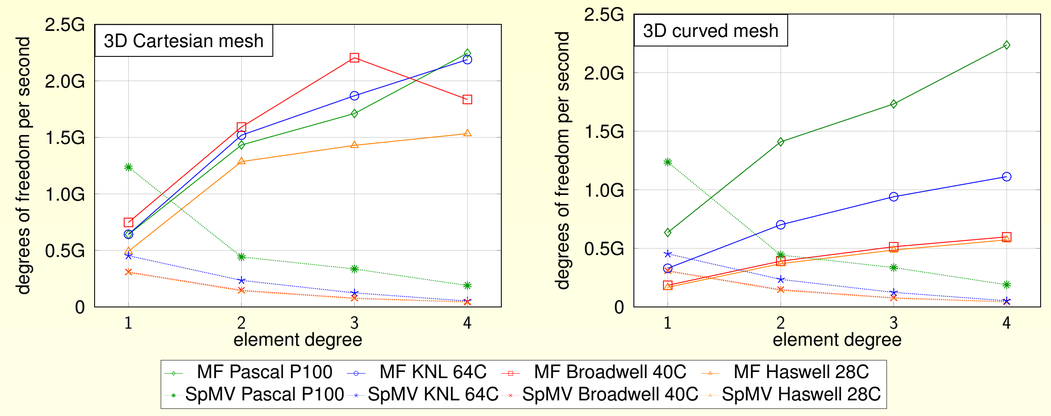
- Laplacian with variable coefficient $a(\mathbf{x}) = 1 + 10^6 \prod_{e=1}^{d} \cos(2\pi x_e + 0.1e)$
- 3D shell geometry, high-order curved elements
- Conjugate gradient iterative solver preconditioned by geometric multigrid based on `deal.II` infrastructure [4, 5], ~ 15 iterations
- Polynomial Chebyshev smoother of degree 5 (=5 mat-vec) for pre- and post-smoothing
- Multigrid cycle done in single precision, outer CG in double precision $\rightarrow$ leverages $2\times$ higher throughput of `float`



Node level results and comparison

Evaluation of Laplacian $(\nabla\varphi, a\nabla u_h)_\Omega$ for continuous $\mathcal{Q}_1$ to $\mathcal{Q}_4$ elements in 3D

- **P100**: One NVIDIA Pascal P100 GPU (inside DGX-1)
- **KNL**: Intel Knights Landing 7210F with 64 cores @ 1.3 GHz; Omnipath interconnect
- **Broadwell**: Intel Xeon E5-2698 v4 server CPU with 2×20 cores @ 2.2 GHz
- **Haswell**: Intel Xeon E5-2697 v3 server CPU with 2×14 cores @ 2.6 GHz



Matrix-free algorithms (MF) more than 10 times faster than sparse-matrix based algorithms (SpMV) $\rightarrow$ **fastest choice** for quadratic and higher order elements

Performance metrics: 3D Cartesian and curved meshes

		GFLOP/s				GB/s			
geometry	degree	Haswell	Broadwell	KNL	P100	Haswell	Broadwell	KNL	P100
Cartesian	1	127	192	166	165	31	47	40	100
Cartesian	2	193	239	228	215	78	92	94	96
Cartesian	3	190	293	250	227	79	102	105	79
Cartesian	4	205	244	291	298	80	93	106	90
curved	1	84	92	165	317	115	125	224	439
curved	2	93	98	177	356	107	115	203	412
curved	3	100	105	193	356	98	104	190	375
curved	4	110	115	215	431	96	101	187	379

P100 $2.0\times$ **faster** than KNL, **KNL** $1.9\times$ **faster** than 2-socket **Broadwell** and **Haswell** on curved mesh

Parallelization techniques and code optimization

Implementation for Intel Xeon and Knights Landing

- Vectorization over several elements by C++ wrapper classes around intrinsics [3] $\rightarrow$ more than 98% of instructions packed
- Haswell and Broadwell use AVX-2 and FMA, KNL uses AVX-512 vectorization
- Developed new intrinsics for gather and scatter access to the global input and output vectors of continuous finite elements with matrix P_K
- Parallelization scheme uses MPI only $\rightarrow$ faster than shared memory where complex sychronization between threads is necessary [2, 3]
- Code compiled with `g++` version 6.3 and Intel MPI 2017
- Programs on KNL use $2\times$ hyperthreading, i.e., 128 ranks on 64 cores, and use fast memory through `mpirun -n 128 numactl --membind=1 ./executable`

Implementation on NVIDIA GPUs

- Separate code path adapted to the data structures of GPU based on CUDA [5]
- Similar interfaces to CPU code, exchange CPU and GPU by changing `typedef`
- Loop over cell parallelized, use fast atomics when accumulating integrals into global vector to avoid race conditions when going through matrix P_K
- Parallelizing over elements does not provide enough parallelism $\rightarrow$ must **parallelize within elements**
  - One thread (Cuda block) per DoF, `__syncthreads()` between tensor product kernels
  - Helps to fit the whole element kernel in registers; similarly to L1 cache on CPU
- GPU implementation optimized for coalesced memory access (structure-of-array)
- GPU code uses specific code for apply hanging node constraints on adaptively refined meshes and multigrid operations [5]

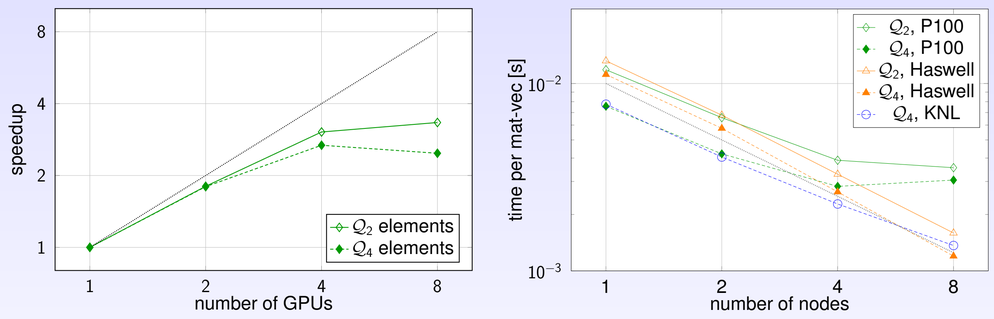
Multi-GPU Setup and Scaling on NVIDIA DGX-1

Layout of multi-GPU algorithm: Similar to MPI parallelization in CPU case [2]

- For all pairs of devices: start $N_{\text{gpu}}/2$ non-blocking communication on $N_{\text{gpu}}/2$ parallel streams using `cudaMemcpyPeerAsync`
- Wait for all streams to be done with their communication calls
- Perform computations in standard way using cell loop

Preliminary work on utilizing several of the eight GPUs in an NVIDIA DGX-1 shows decent scaling up to four devices, provided the problem size is not too small

Multi-GPU scaling on 3D Cartesian mesh case with 17m DoFs



- CPU version scales ideally, including slight superlinear improvement between 2 and 4 nodes when whole kernel fits in L3 cache
- GPU scaling saturates at around 3 ms
- Techniques for improving scalability currently under investigation: efficient usage of CUDA streams, more advanced communication patterns inspired by existing efforts for distributed MPI computations

References

[1] B. Krank, N. Fehn, W. A. Wall, M. Kronbichler (2017), A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *Journal of Computational Physics*. `doi:10.1016/j.jcp.2017.07.039`.

[2] M. Kronbichler, K. Kormann (2012), A generic interface for parallel cell-based finite element operator application, *Computers & Fluids* 63:135–147. `doi:10.1016/j.compfluid.2012.04.012`.

[3] M. Kronbichler, K. Kormann, I. Pasichnyk, M. Allalen: Fast Matrix-Free Discontinuous Galerkin Kernels on Modern Computer Architectures, in J. Kunkel , R. Yokota, P. Balaji, D. Keyes (eds): *High Performance Computing. ISC 2017*. Lecture Notes in Computer Science, vol 10266, pp. 237–255 (2017), `doi:10.1007/978-3-319-58667-0_13`.

[4] M. Kronbichler, W. A. Wall (2016), A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, arXiv preprint arXiv:1611.03029.

[5] K. Ljungkvist, M. Kronbichler (2017), Multigrid for Matrix-Free Finite Element Computations on Graphics Processors, Technical Report 2017-006, Department of Information Technology, Uppsala University.