Deep Learning Hardware Accelerates Fused Discontinuous Galerkin Simulations

Introduction, Scope, Summary

- In recent years the compute/memory balance of processors has been continuously shifting towards compute
- The rise of Deep Learning, which is based on matrix multiplication, accelerated this path, especially in terms of single precision (FP32) and lower precision compute
- An important research question is if this development can be leveraged for traditional HPC
- We demonstrate that a high order discontinuous Galerkin solver for seismic wave equations can execute in single precision without any loss of modeling accuracy when running application scenarios
- We extended our solver to support the Intel Knights Mill CPU with 14 TFLOPS of single precision deep-learning performance in its small sparse matrix-matrix kernels
- Compared to the HPC-focused Knights Landing processor speed-ups of up to 1.6× are possible depending on the scenario
- Knights Mill is therefore even able to match dual socket top-bin Xeon performance in case of FP32 execution

Kernels

For an efficient execution, two sparse kernels need to be accelerated by hard-wiring the sparsity patterns using runtime code generation:

- *K1*: sparse-matrix × 3D-tensor = 3D-tensor, this operation is needed for multiplication with Jacobians and flux-solvers. In BLAS-notation, the sparse matrix A is a 9×9 matrix, whereas B and C are dense 3D-tensors.
- *K2*: 3D-tensor × sparse-matrix = 3D-tensor, this operation is needed for multiplication with stiffness or flux matrices. The dimensions of the sparse matrix B depend on the order and stage of the integration kernels.

Generator sketch of kernel *K1*:

```
1:  $nb \leftarrow [\#\text{modes}/\text{scratchpad\_size}]$
2:  for all $blk = 1$ to $nb$ do
3:    for all $m = 1$ to #quantities do
4:      $a_{\#\text{Entries}} \leftarrow \text{row}_A[m+1] - \text{row}_A[m]$
5:      for $k = 1$ to $a_{\#\text{Entries}}$ do
6:        $a[1:f] \leftarrow \text{broadcast}(A[\text{row}_A[m]+k])$
7:        for all $n = (blk-1) \cdot \text{scratchpad\_size}$ to $blk \cdot \text{scratchpad\_size}$ do
8:          $b[1:f] \leftarrow B[\text{col}_A[\text{row}_A[m]+k]][n][1:f]$
9:          $C[m][n][1:f] \leftarrow \text{FMA}(a[1:f], b[1:f], C[m][n][1:f])$
10:       end for
11:     end for
12:   end for
13: end for
```

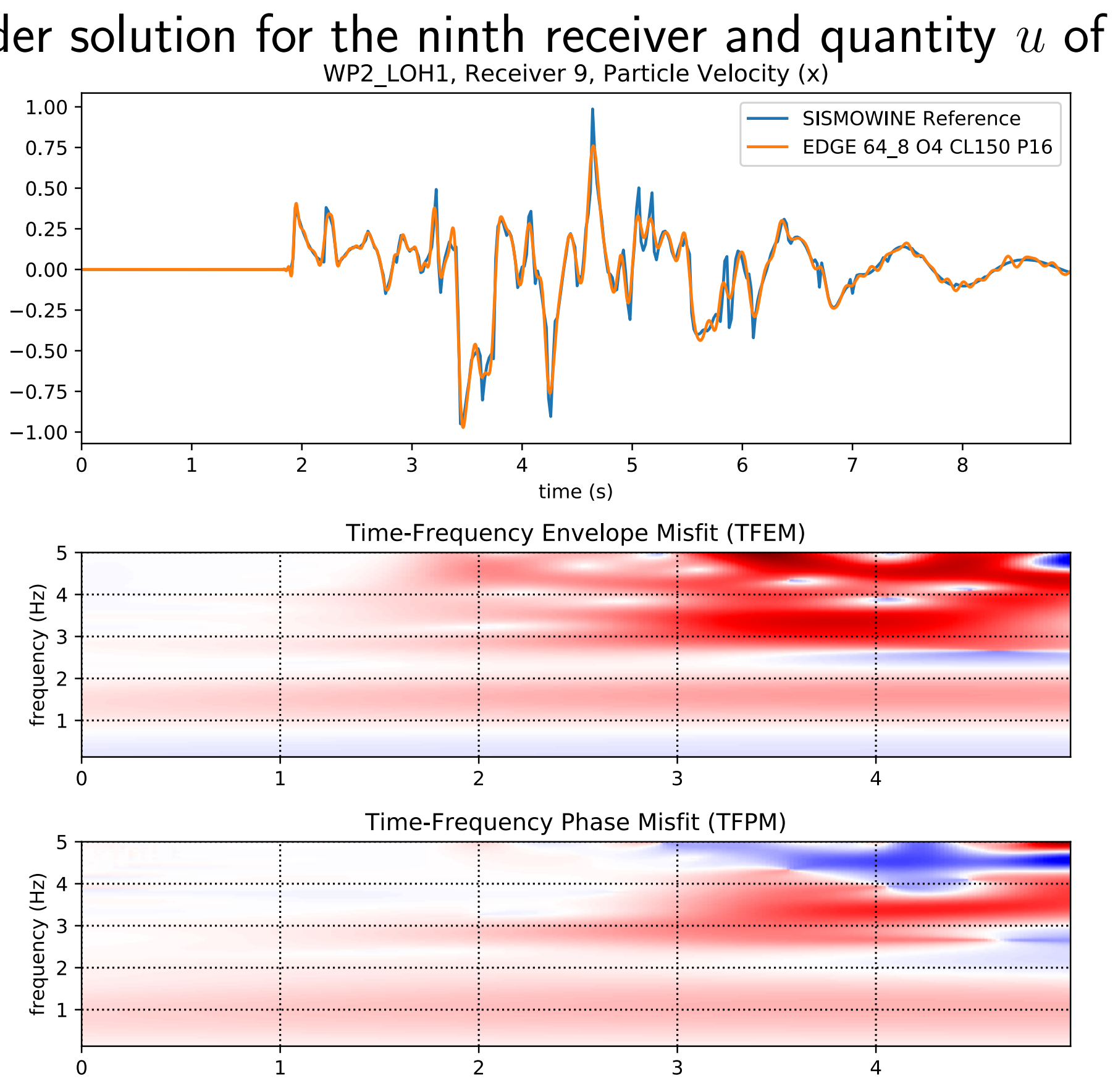
Generator sketch of kernel *K2*:

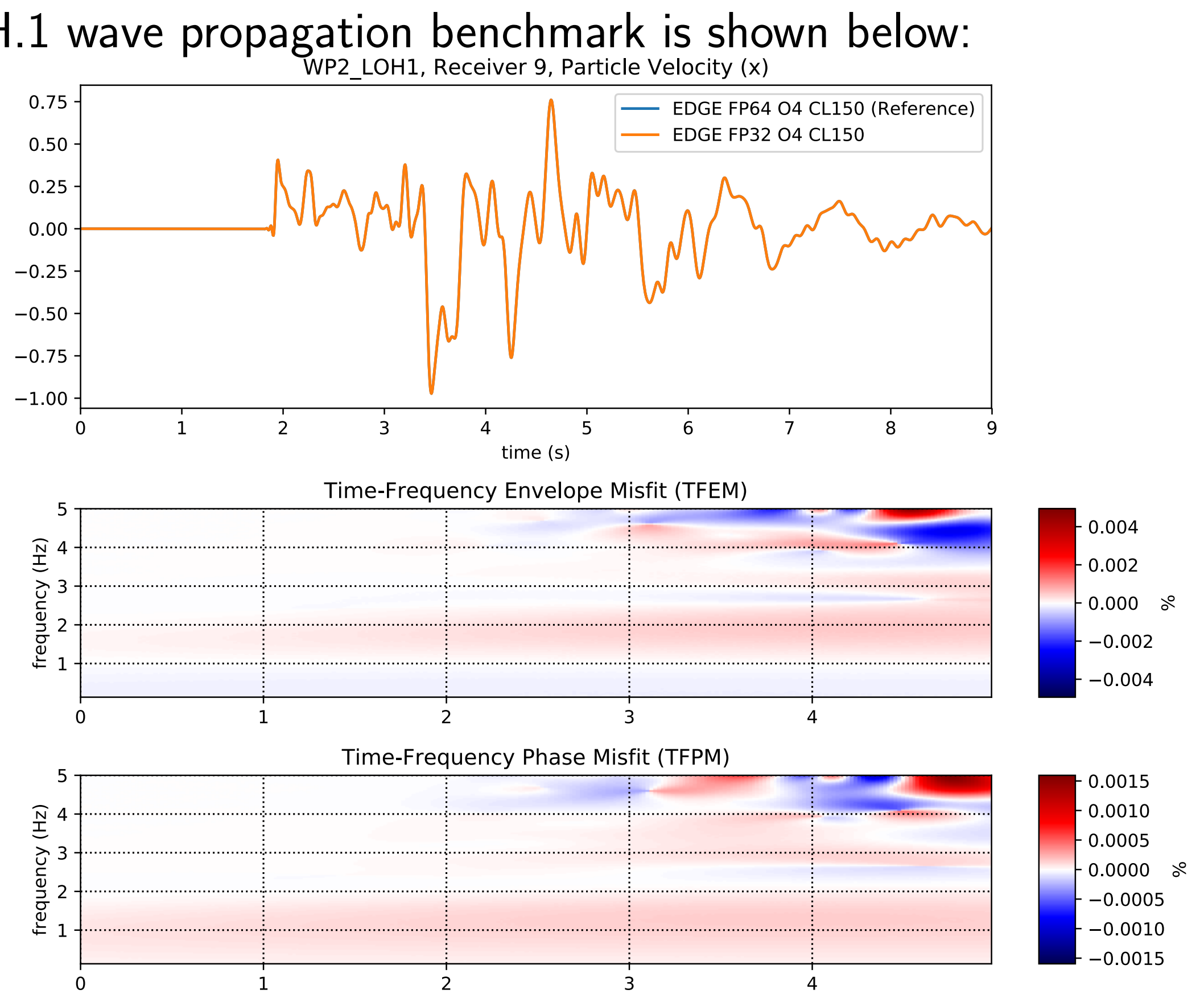
```
1:  $nb \leftarrow [\#\text{modes}/\text{scratchpad\_size}]$
2:  for all $m = 1$ to #quantities do
3:    for all $blk = 1$ to $nb$ do
4:      $n_0 \leftarrow (blk-1) \cdot \text{scratchpad\_size}$
5:      for all $n = 1$ to scratchpad_size do $c_n[1:f] \leftarrow C[m][n_0+n][1:f]$ end for
6:      for all $k = 1$ to #modes do
7:        for all $n = 1$ to scratchpad_size do
8:          $b_{\#\text{Entries}} \leftarrow \text{col}_B[n_0+n+1] - \text{col}_B[n_0+n]$
9:          for $l = 1$ to $b_{\#\text{Entries}}$ do
10:           if $\text{row}_B[\text{col}_B[n_0+n]+l] == k$ then
11:             $b[1:f] \leftarrow \text{broadcast}(B[\text{col}_B[n_0+n]+l])$
12:             $c_n[1:f] \leftarrow \text{FMA}(A[m][k][1:f], b[1:f], c_n[1:f])$
13:           end if
14:         end for
15:       end for
16:       for all $n = 1$ to $bksz_n$ do $C[m][n_0+n][1:f] \leftarrow c_n[1:f]$ end for
17:     end for
18:   end for
19: end for
```

Governing Equations and Numerical Results for FP32 vs. FP64

The three-dimensional isotropic elastic wave equations in velocity-stress formulation are given as a system of linear hyperbolic partial differential equations. As shown in previous work these equations can be solved as series of small sparse matrix matrix products when applying the ADER-DG machinery for variable convergence rate. Additionally, we leverage the fact that many of the grand challenges in earthquake system science require large ensembles of geometrically similar forward simulations. Our solver S_m operates in parallel on $m \leq n$ different inputs $I_m = (i_1, i_2, \ldots, i_m)$ to obtain a set of observations in a single execution: $O_m = (o_1, o_2, \ldots, o_m) = S_m(I_m)$. This simple idea of fusing m simulations is the basic paradigm in our software. The advantages of this approach range from higher data-reuse through shared data structures, e.g., the mesh or velocity model, towards better parallelization opportunities at all levels, as each element in the mesh is represented as a 3d-tensor: modes × quantities × fused runs.

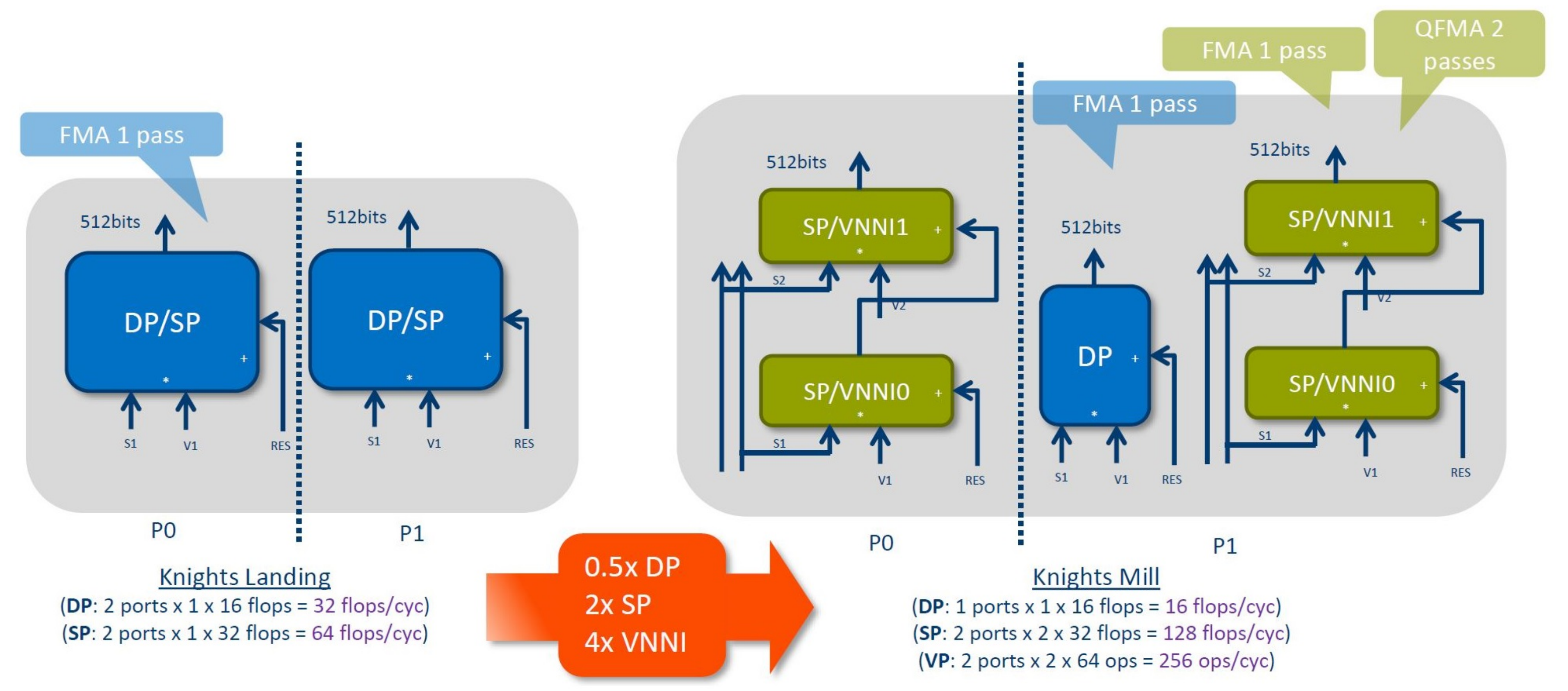
We executed for several application benchmarks (HHS1, HSP1a, HSP1b, LOH.1) convergence rates $\mathcal{O} \in \{2, \ldots, 7\}$ in single (FP32) and double precision (FP64). We only observed negligible misfits and can conclude that single precision arithmetic is a sufficient for our wave propagation solver. In total, this led in case of LOH.1 to $6 \times 2 \times 9 \times 3 = 324$ synthetic seismograms for the six orders, two precisions, nine receivers, and three velocity components. An exemplary illustration of our solvers fourth order solution for the ninth receiver and quantity u of the LOH.1 wave propagation benchmark is shown below:





Leveraging AVX512 Single Precision Deep Learning Oriented Hardware

Knights Landing (knl) vs. Knights Mill (knm) VPU: a symmetric, single-pumped combo VPU is replaced by an asymmetric (single precision biased) VPU which is double-pumped for high efficiencies on the two-issue wide Xeon Phi frontend. The chained double-pumped unit can be used by the so-called 4FMA instruction which implements a matrix vector multiplication, $M = 16, N = 1, K = 4$.

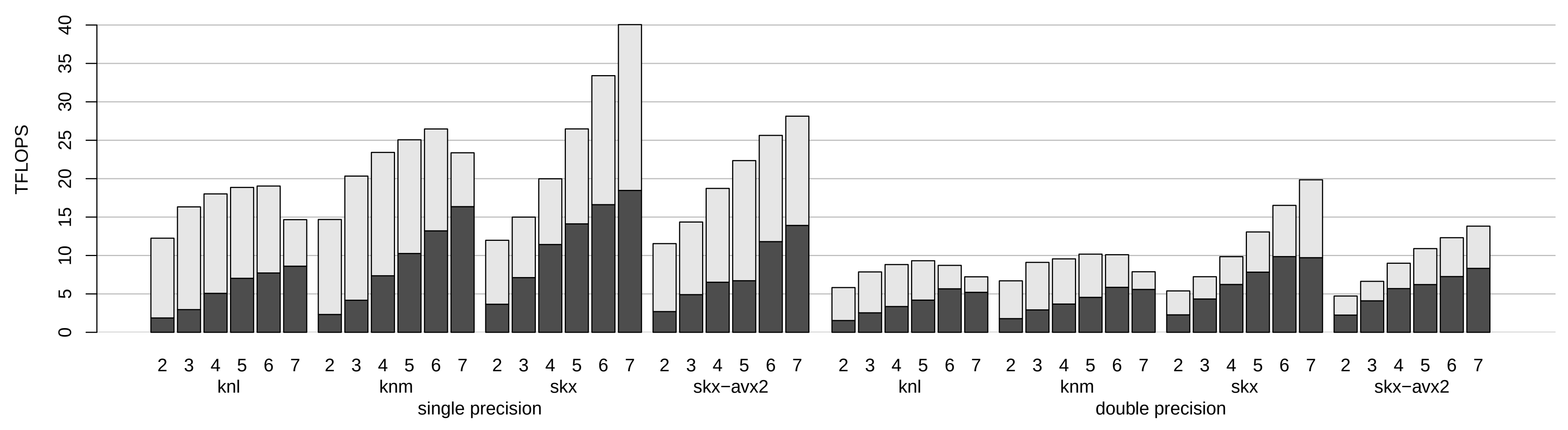


FMA 1 pass
512bits
DP/SP
S1
V1
RES
P0
P1
Knights Landing
(DP: 2 ports x 1 x 16 flops = 32 flops/cyc)
(SP: 2 ports x 1 x 32 flops = 64 flops/cyc)
0.5x DP
2x SP
4x VNNI
FMA 1 pass
QFMA 2 passes
SP/VNNI1
SP/VNNI0
DP
S2
V2
Knights Mill
(DP: 1 ports x 1 x 16 flops = 16 flops/cyc)
(SP: 2 ports x 2 x 32 flops = 128 flops/cyc)
(VP: 2 ports x 2 x 64 ops = 256 ops/cyc)

Only kernel *K2* (which consumes most flops) can be potentially accelerated by 4FMA instructions. The detection of a possible 4FMA instruction is carried out by modifying the check in line 10 of kernel *K2*. Instead of only checking for the current k when iterating over the modes of a specific quantity, this check is extended as follows: if $\text{row}_B[\text{col}_B[n_0+n]+l] == k$, $\text{row}_B[\text{col}_B[n_0+n]+l+1] == k+1$, $\text{row}_B[\text{col}_B[n_0+n]+l+2] == k+2$ and finally $\text{row}_B[\text{col}_B[n_0+n]+l+3] == k+3$ are all true, a 4FMA instruction can be issued. By applying this algorithm several times we can add zero fill-in which increases the performance by up to 10%.

Multinode Application Performance

We have evaluated the performance on 16 nodes of Intel Xeon Phi 7250 (knl), 16 nodes of dual-socket Xeon Platinum 8180 (SKX, for AVX512 and AVX2), and 16 nodes of Intel Xeon Phi 7295 (knm), all connected by Intel Omni-Path. For higher orders of convergence the shared L2 cache on knl and knm becomes a bottleneck due to JIT code size, however FP64 knl and knm deliver the same performance as knl is limited by its narrow frontend. For orders up to $\mathcal{O} = 5$, knm can match the absolute FP32 performance of dual socket Xeon (much higher power consumption) using 4FMA instructions. Compared to knl, knm achieves up 1.6X speed-up over knl for the same reason.


TFLOPS
knl
knm
skx
skx−avx2
single precision
double precision