Tensor-Optimized Hardware Accelerates Fused Discontinuous Galerkin Simulations

Alexander Breuer*, Alexander Heinecke† and Yifeng Cui*

*San Diego Supercomputer Center, UC San Diego, San Diego, CA, USA
†Parallel Computing Lab, Intel Corporation, Santa Clara, CA, USA

***Abstract*—In recent years the computation/memory balance of processors continuously shifted towards computation. The rise of Deep Learning, which is based on matrix multiplications, accelerated this path, especially in terms of single precision and lower precision computation. An important research question is if this development can be leveraged for traditional HPC. In this work we demonstrate that a high order discontinuous Galerkin solver for seismic wave propagation can execute in single precision without loss of modeling accuracy. Additionally, we extended its kernels to support the Intel Knights Mill CPU with 14 TFLOPS of tensor-optimized single precision performance. This allows us to exploit the hardware's special computation capabilities, even in a regular HPC application with sparse linear algebra kernels. At the cluster-level, Knights Mill can obtain the same application performance as the latest top-bin dual socket Intel Xeon Platinum nodes. Compared to the HPC-focused Knights Landing processor, speed-ups of up to 1.6× are possible, depending on the scenario. Additionally, we are able to increase the throughput of seismic discontinuous Galerkin methods by 4.2×, when comparing our solver's single precision and fifth order performance to the SC 2017 best-paper award winning work.**

I. Introduction And Related Work

Until two years ago, the development of processors for traditional High Performance Computing (HPC) use cases has targeted a more and more complicated design principle: "Increase the memory bandwidth at the same rate as the compute capabilities of the processor itself". The need for a balanced platform is based on the algorithmic patterns of HPC workloads. The fast rise of deep learning changed this design view of computer architectures. Deep learning algorithms rely on tensor operations, implemented as General Matrix-Matrix Multiplications (GEMMs). More importantly, single precision (FP32) GEMM, half precision (FP16), or even quantized integers are used for deep learning. Therefore, several vendors invested in developing processors with focus on lowered precision, while not increasing memory bandwidth.

We examine how the seismic solver EDGE [1] can benefit from less-balanced, tensor-optimized hardware. In addition, we compare our obtained performance to regular general purpose processors with similar machine balances but large caches and wide execution engines. Optimal utilization of tensor-optimized hardware by EDGE required several steps, which also helps with best performance on general purpose CPUs. As illustrated in Fig. 1, our poster's contribution is two-fold: a) applying, analyzing and verifying algorithmic advances to EDGE for general purpose CPUs, b) discussion and evaluation of optimizations needed for tensor-optimized hardware. Our poster reports on the following contributions:

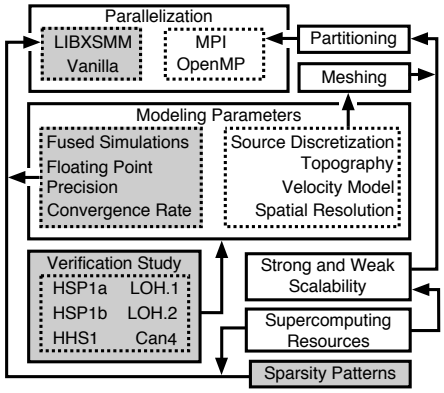

Fig. 1. Illustration of EDGE's modeling and simulation pipeline. The parallelization is influenced by a multitude of factors. Our areas of contribution are highlighted in gray.

- Extension of previous fused DG-FEM simulation technology to arbitrary orders of convergence and high-performance x86 CPU architectures.
- Analysis of FP32 accuracy and performance in an extensive verification study for seismic wave propagation. The analysis of the LOH.1 benchmark is part this work, including an electronic supplement and references to further benchmark results (see App. A).
- Revised Just-In-Time (JIT) assembly kernel generation of core computational routines in the LIBXSMM library, targeting the various processors of our study. Tensor-optimized hardware, which is very specific to dense GEMM, allows us to outperform the SC 2017 best-paper award winning work [2] by $4.2\times$.

II. Fused Seismic Simulations

EDGE solves the three-dimensional isotropic elastic wave equations in velocity-stress formulation. These are given as a system of linear hyperbolic partial differential equations:

$$q_t + Aq_x + Bq_y + Cq_z = S. \tag{1}$$

t is time and $\vec{x} = (x, y, z)$ the vector of Cartesian coordinates. Subscripts in Eq. 1 denote partial derivatives. $q(\vec{x}, t) = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{xz}, \sigma_{yz}, u, v, w)$ is the space-time-dependent vector of quantities, containing the three nor-

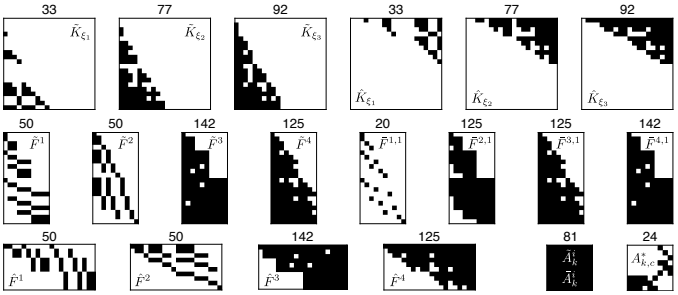

Fig. 2. Sparsity patterns of the ADER-DG matrices for a fourth order method. On top of each matrix, the number of non-zero entries is given.

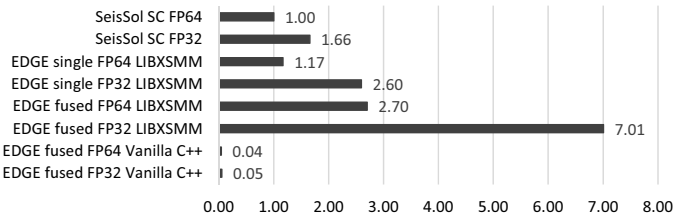

Fig. 3. Speed-up over the SC 2017 best paper award-winning work [2] on the KNM processor using fifth order and global time stepping.

mal stresses σ_{xx}, σ_{yy} and σ_{zz}, the three shear stresses σ_{xy}, σ_{xz}, and σ_{yz}, and the three particle velocities u, v and w. $A(\vec{x})$, $B(\vec{x})$, $C(\vec{x})$ are the three space-dependent Jacobians, discretizing the seismic velocity model. Application of the ADER-DG machinery leads to a scheme, composed of sparse matrix-matrix operations [1]. These matrices are repeatedly applied to the degrees of freedom to advance the simulation in time and illustrated for a fourth order method in Fig. 2.

EDGE heavily exploits the fact that many of the grand challenges in earthquake system science require large ensembles of geometrically similar forward simulations. A traditional solver s operates on fixed input i to obtain observations $o = s(i)$. Now, if we require observations for n different inputs $I_n = (i_1, i_2, \ldots, i_n)$, e.g., different seismic sources, we would execute the solver n times to obtain the set of observations $O_n = (o_1, o_2, \ldots, o_n) = (s(i_1), s(i_2), \ldots, s(i_n))$. In contrast, EDGE's 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: $O_m = (o_1, o_2, \ldots, o_m) = S_m(I_m)$. The advantages of this basic paradigm range from higher data-reuse through shared data structures, e.g., the velocity model, towards better parallelization opportunities at all levels.

Single vs. Double Precision Floating Point Arithmetic: In the past, seismic ADER-DG simulations have been carried out in double precision arithmetic [2]–[6]. The reasons are: a) extensive verification was only performed for double precision, and b) potential speedups are annihilated through low SIMD parallelism of non-fused ADER-DG kernels. As part of a larger parameter study, c.f. App. A, we present verification results using the Layer Over Halfspace benchmark. We executed the benchmark for convergence rates $\mathcal{O} \in \{2, \ldots, 6\}$ in single and double precision arithmetic. We observe a good fit of EDGE's solutions with the reference, while the 32-bit synthetics are visually indistinguishable from the 64-bit solution (see App. A). In conjunction with our extensive multi-parameter study and other verification settings [7], we conclude that single precision arithmetic is sufficient for EDGE's wave propagation solver. This observation agrees with other seismic solvers, e.g., [8] or [9].

III. Kernel Optimizations

For the presented work we extended LIBXSMM heavily. This extension is many-fold: a) small dense regular GEMM with QFMA (Knight Mill) support, b) small dense fused GEMM with QFMA (Knights Mill) support (for fused simulation with low sparsity, e.g. the flux solver), c) sparse fused GEMM with QFMA (Knights Mill) support, d) small dense fused GEMM for regular AVX2 and AVX512 architectures e) sparse fused GEMM for regular AVX2 and AVX512 architectures. Our optimizations hardwire the sparsity pattern of the sparse operator matrices to maximize execution of non-zero FLOPs. That means, runtime-compiling an assembly kernel into the instruction stream that exactly matches the sparse matrix operator. We optimized for low overheads stemming from processing the sparse matrix structure and selecting the corresponding DOF-matrix entries at runtime. A pseudo-code example for c) is given in App. A.

Improvement Over State-Of-The-Art: Fig. 3 depicts the speedups over the SC 2017 best paper award-winning work [2] on the Knights Mill processor using order $\mathcal{O} = 5$[1]. This is done by using a single node setup of the LOH.1 benchmark with 350,264 elements. Due to our JIT'ed assembly kernels we can outperform the SC 2017 work. This even is true without leveraging fused simulations or single precision arithmetic. The combined speedup of EDGE over [2] is 7.0×, when exploiting FP32 and fused simulations as modeling parameters (see Fig. 1). When also running SeisSol in single precision, EDGE's fused simulation throughput is still $4.2\times$ higher. Since [2] only uses LIBXSMM in a static way, but not at runtime, SeisSol cannot make use of QFMA in case of the FP32 execution. Fig. 3 also highlights that our fused assembly kernels utilize the hardware much more efficiently than single runs or [2] because all multiplications with padded zeros disappear and perfect vector code is executed. Additionally, we can see why runtime code generation tailored to the sparse matrix operators is essential. The performance delivered by running EDGE using the Intel compiler with vectorization pragmas (denoted by C++) is extremely poor, although it executes fused simulations. Our runtime assembly kernel generators, that have been added to LIBXSMM, are able to outperform the compiler by 140x (FP32) and 65x (FP64) as the compiler fails to vectorize the dense C++ vanilla kernels, but also misses essential prefetching [10] and blocking strategies for ADER-DG.

[1]We compiled SeisSol according to [2]'s artifact description and confirmed that our binary reproduces the reported performance on the same hardware.

REFERENCES

[1] A. Breuer, A. Heinecke, and Y. Cui, "Edge: Extreme scale fused seismic simulations with the discontinuous galerkin method," in *International Supercomputing Conference*. Springer, 2017, pp. 41–60.

[2] C. Uphoff, S. Rettenberger, M. Bader, E. H. Madden, T. Ulrich, S. Wollherr, and A.-A. Gabriel, "Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake," 2017.

[3] A. Breuer, A. Heinecke, and M. Bader, "Petascale local time stepping for the ader-dg finite element method," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, 2016.

[4] A. Heinecke, A. Breuer, M. Bader, and P. Dubey, "High order seismic simulations on the intel xeon phi processor (knights landing)," in *International Conference on High Performance Computing*, 2016.

[5] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan *et al.*, "Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[6] A. Breuer, A. Heinecke, S. Rettenberger, M. Bader, A. Gabriel, and C. Pelties, "Sustained petascale performance of seismic simulations with SeisSol on SuperMUC," in *International Supercomputing Conference*, 2014.

[7] Alexander Breuer, Alexander Heinecke, Yifeng Cui, "EDGE: Benchmarking the Seismic Wave Propagation Solver," in *Proceedings of the 11th National Conference in Earthquake Engineering, Earthquake Engineering Research Institute, Los Angeles, CA. 2018 (to appear)*.

[8] H. Fu, C. He, B. Chen, Z. Yin, Z. Zhang, W. Zhang, T. Zhang, W. Xue, W. Liu, W. Yin *et al.*, "18.9-pflops nonlinear earthquake simulation on sunway taihulight: enabling depiction of 18-hz and 8-meter scenarios," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 2.

[9] M. Rietmann, P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini, "Forward and adjoint simulations of seismic wave propagation on emerging large-scale gpu architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 38.

[10] "Ch. 21," in *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*, J. Reinders, J. Jeffers, and A. Sodani, Eds., 2016.

A. Artifact Description and Evaluation Appendix: Tensor-Optimized Optimized Hardware Accelerates Fused Discontinuous Galerkin Simulations

A. Abstract

This artifact description appendix gives an overview of the used software and hardware. We provide details on the availability of all components and describe details of the solver EDGE and library LIBXSMM. This description is complemented by electronic supplements, providing a multitude of configurations for the Layer Over Halfspace benchmark 1 (LOH.1). Further, we describe the experiment workflow, give details on how performance was measured.

B. Description

1) Check-list (Artifact meta information):

- **Algorithm:** Arbitrary high order DERivatives Discontinuous Galerkin Finite Element Method (ADER-DG-FEM) solving the elastic wave equations in velocity-stress form.
- **Program:** EDGE (solver), LIBXSMM (kernels).
- **Data set:** Discretization of the LOH.1 benchmark with an unstructured tetrahedral mesh (1,864,616 elements) and a double-couple point source.
- **Hardware:** Intel Xeon Phi 7250, Intel Xeon Phi 7295, Intel Scalable Xeon 8180.
- **Execution:** MPI+OpenMP.
- **Output:** Nine receivers at the free-surface boundary (verification runs only).
- **Experiment workflow:** 1) Pre-processing: Meshing, partitioning and source sampling, 2) Simulation, 3) Comparison with reference solution (verification only).
- **Publicly available?:** Yes, BSD3 for software, CC0'd data.

2) How software can be obtained: EDGE is available from http://dial3343.org (BSD3). LIBXSMM is available from https://github.com/hfp/libxsmm (BSD3).

3) Hardware dependencies: Optimized, fused kernels are available in single (8 or 16 simulations) and double precision arithmetic (4 or8 simulations) for architectures supporting AVX, AVX2, AVX512 or AVX512+QFMA. Optimized kernels for single forward simulations are available for architectures supporting SSE3, AVX, AVX2, or AVX512. As a fallback, vanilla kernels, not using LIBXSMM, are available.

4) Datasets: As part of this work, we provide input and output of a verification study for the LOH.1 benchmark. The input covers 10 different configurations with varying refinement. We share the Gmsh-scripts, generated ASCII meshes, partitioned HDF5-meshes, the sampled point-source, and EDGE's XML-configurations. Raw and analyzed output is available for 6 inputs, using single and double precision, and orders 2-7 (see Artifact Evaluation). We provide raw and analyzed results for six different mesh resolutions, six different orders, and single and double precision arithmetic. All input data, output data, and scripts of our LOH.1 verification study are available from http://doi.org/10.17605/OSF.IO/H9G5N. Data of further verification benchmarks is available from EDGE's data repository at http://opt.dial3343.org.

	$\mathcal{O}=2$	$\mathcal{O}=3$	$\mathcal{O}=4$	$\mathcal{O}=5$	$\mathcal{O}=6$
sparse	6,642	19,944	52,002	121,032	260,370
dense	8,064	30,888	98,136	267,336	643,680
overhead	21 %	55 %	89 %	121 %	147 %

TABLE I
Required number of floating point operations per element update of the ADER-DG scheme in dependence of the convergence rate. Shown are sparse and dense matrix operators. The last row shows the respective overhead.

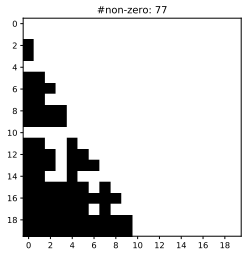

Fig. 4. Sparsity pattern of the second transposed stiffmatrix in fourth order.

C. Evaluation and Expected Result

For all multi-node application performance measurements, we used the partitioned mesh provided in the archive `tet4_150_16.h5m.tar.bz2` of the electronic artifacts. The respective configuration is given in `tet4_150.xml`, where we disabled the receivers and reduced the simulation's end time for a target execution time of at least 20 minutes in all configurations. EDGE prints the time consumption of the computational loop over the time steps. We used this timer and the reported number of floating point operations in Tab. I to derive the respective non-zero performance reported in this work.

D. LIBXSMM Kernel Generation

Efficient utilization of sparse fused GEMM with QFMA (Knights Mill), required us to optimize two kernels:

- *K1*: sparse-matrix $\times$ 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 $\times$ 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.

Pseudo-code for K2 in Alg. 1. According to [1] the sparse matrix should be stored in compressed sparse row (CSR) format, to match the row major storage format of the 3D-tensors. We highlight the LIBXSMM-generated KNM code by using the second transposed stiffness matrix in fourth order, which falls into the *K2* category. Its sparsity pattern is depicted in Fig. 4. The runtime-generated code for 16 fused runs on KNM for FP32 of the LIBXSMM generator implemented in this work is given below. The assembly kernel obeys to System-V x86_64 ABI, which means that the matrix's A pointer comes in `rsi`, matrix's B is provided through `rdi` and matrix's C by `rdx`. The above shown extremely

Algorithm 1 Code generator sketch of kernel *K2*, sparse matrix B is stored in CSC format.

$nb \leftarrow \lceil \#\text{modes}/\text{scratchpad\_size} \rceil$
for all $m = 1$ to #quantities **do**
  for all $blk = 1$ to nb **do**
    $n_0 \leftarrow (blk - 1) \cdot \text{scratchpad\_size}$
    for all $n = 1$ to scratchpad_size **do** $c_n[1:f] \leftarrow C[m][n_0 + n][1:f]$ **end for**
    for all $k = 1$ to #modes **do**
      for all $n = 1$ to scratchpad_size **do**
        $b_{\#\text{Entries}} \leftarrow \text{col}_B[n_0 + n + 1] - \text{col}_B[n_0 + n]$
        for $l = 1$ to $b_{\#\text{Entries}}$ **do**
          if $\text{row}_B[\text{col}_B[n_0 + n] + l] == k$ **then**
            $b[1:f] \leftarrow \text{broadcast}(B[\text{col}_B[n_0 + n] + l])$
            $c_n[1:f] \leftarrow \text{FMA}(A[m][k][1:f], b[1:f], c_n[1:f])$
          end if
        end for
      end for
      for all $n = 1$ to $bksz_n$ **do** $C[m][n_0+n][1:f] \leftarrow c_n[1:f]$ **end for**
    end for
  end for
end for

high efficiency is now obvious: the kernel only consists of fully-vectorized AVX512 instruction and not a single scalar instruction in the critical loop. `r12` is used to loop over the quantities. `r13,r14,r15` are not hot in this particular kernel.

```
push rbx
push r12
push r13
push r14
push r15
mov r12, 0x0
mov r13, 0x0
mov r14, 0x0
0x1e:
add r12, 0x1
vmovups zmm0, zmmword ptr [rdx]
vmovups zmm1, zmmword ptr [rdx+0x40]
vmovups zmm2, zmmword ptr [rdx+0x80]
vmovups zmm3, zmmword ptr [rdx+0xc0]
vmovups zmm4, zmmword ptr [rdx+0x100]
vmovups zmm5, zmmword ptr [rdx+0x140]
vmovups zmm6, zmmword ptr [rdx+0x180]
vmovups zmm7, zmmword ptr [rdx+0x1c0]
vmovups zmm8, zmmword ptr [rdx+0x200]
vmovups zmm9, zmmword ptr [rdx+0x240]
vmovups zmm10, zmmword ptr [rdx+0x280]
vmovups zmm11, zmmword ptr [rdx+0x2c0]
vmovups zmm12, zmmword ptr [rdx+0x300]
vmovups zmm13, zmmword ptr [rdx+0x340]
vmovups zmm14, zmmword ptr [rdx+0x380]
vmovups zmm15, zmmword ptr [rdx+0x3c0]
vmovups zmm16, zmmword ptr [rdx+0x400]
vmovups zmm17, zmmword ptr [rdx+0x440]
vmovups zmm18, zmmword ptr [rdx+0x480]
vmovups zmm19, zmmword ptr [rdx+0x4c0]
vmovups zmm28, zmmword ptr [rdi]
vmovups zmm28, zmmword ptr [rdi+0x40]
vmovups zmm28, zmmword ptr [rdi+0x80]
vfmadd231ps zmm0, zmm28, dword ptr [rsi]{1to16}
vmovups zmm28, zmmword ptr [rdi+0xc0]
vfmadd231ps zmm0, zmm28, dword ptr [rsi+0x4]{1to16}
vmovups zmm28, zmmword ptr [rdi+0x100]
vmovups zmm28, zmmword ptr [rdi+0x140]
vmovups zmm29, zmmword ptr [rdi+0x180]
vmovups zmm30, zmmword ptr [rdi+0x1c0]
vmovups zmm31, zmmword ptr [rdi+0x200]
v4fmaddps zmm0, zmm28, xmmword ptr [rsi+0x8]
v4fmaddps zmm1, zmm28, xmmword ptr [rsi+0x40]
vfmadd231ps zmm2, zmm29, dword ptr [rsi+0x78]{1to16}
vfmadd231ps zmm2, zmm31, dword ptr [rsi+0x7c]{1to16}
vfmadd231ps zmm3, zmm31, dword ptr [rsi+0xa0]{1to16}
vmovups zmm28, zmmword ptr [rdi+0x240]
vfmadd231ps zmm0, zmm28, dword ptr [rsi+0x18]{1to16}
vfmadd231ps zmm1, zmm28, dword ptr [rsi+0x50]{1to16}
vfmadd231ps zmm2, zmm28, dword ptr [rsi+0x80]{1to16}
vfmadd231ps zmm3, zmm28, dword ptr [rsi+0xa4]{1to16}
vmovups zmm28, zmmword ptr [rdi+0x280]
vmovups zmm28, zmmword ptr [rdi+0x2c0]
vmovups zmm29, zmmword ptr [rdi+0x300]
vmovups zmm30, zmmword ptr [rdi+0x340]
vmovups zmm31, zmmword ptr [rdi+0x380]
v4fmaddps zmm0, zmm28, xmmword ptr [rsi+0x1c]
v4fmaddps zmm1, zmm28, xmmword ptr [rsi+0x54]
vfmadd231ps zmm2, zmm29, dword ptr [rsi+0x84]{1to16}
vfmadd231ps zmm2, zmm30, dword ptr [rsi+0x88]{1to16}
v4fmaddps zmm4, zmm28, xmmword ptr [rsi+0xbc]
vfmadd231ps zmm5, zmm29, dword ptr [rsi+0xe0]{1to16}
vfmadd231ps zmm5, zmm30, dword ptr [rsi+0xe4]{1to16}
vfmadd231ps zmm6, zmm30, dword ptr [rsi+0xfc]{1to16}
vmovups zmm28, zmmword ptr [rdi+0x3c0]
vmovups zmm29, zmmword ptr [rdi+0x400]
vmovups zmm30, zmmword ptr [rdi+0x440]
vmovups zmm31, zmmword ptr [rdi+0x480]
v4fmaddps zmm0, zmm28, xmmword ptr [rsi+0x2c]
v4fmaddps zmm1, zmm28, xmmword ptr [rsi+0x64]
v4fmaddps zmm2, zmm28, xmmword ptr [rsi+0x8c]
v4fmaddps zmm3, zmm28, xmmword ptr [rsi+0xa8]
v4fmaddps zmm4, zmm28, xmmword ptr [rsi+0xcc]
v4fmaddps zmm5, zmm28, xmmword ptr [rsi+0xe8]
vfmadd231ps zmm6, zmm29, dword ptr [rsi+0x100]{1to16}
vfmadd231ps zmm6, zmm31, dword ptr [rsi+0x104]{1to16}
v4fmaddps zmm7, zmm28, xmmword ptr [rsi+0x10c]
vfmadd231ps zmm8, zmm29, dword ptr [rsi+0x120]{1to16}
vfmadd231ps zmm8, zmm31, dword ptr [rsi+0x124]{1to16}
vfmadd231ps zmm9, zmm31, dword ptr [rsi+0x12c]{1to16}
vmovups zmm28, zmmword ptr [rdi+0x4c0]
vfmadd231ps zmm0, zmm28, dword ptr [rsi+0x3c]{1to16}
vfmadd231ps zmm1, zmm28, dword ptr [rsi+0x74]{1to16}
vfmadd231ps zmm2, zmm28, dword ptr [rsi+0x9c]{1to16}
vfmadd231ps zmm3, zmm28, dword ptr [rsi+0xb8]{1to16}
vfmadd231ps zmm4, zmm28, dword ptr [rsi+0xdc]{1to16}
vfmadd231ps zmm5, zmm28, dword ptr [rsi+0xf8]{1to16}
vfmadd231ps zmm6, zmm28, dword ptr [rsi+0x108]{1to16}
vfmadd231ps zmm7, zmm28, dword ptr [rsi+0x11c]{1to16}
vfmadd231ps zmm8, zmm28, dword ptr [rsi+0x128]{1to16}
vfmadd231ps zmm9, zmm28, dword ptr [rsi+0x130]{1to16}
vmovups zmmword ptr [rdx], zmm0
vmovups zmmword ptr [rdx+0x40], zmm1
vmovups zmmword ptr [rdx+0x80], zmm2
vmovups zmmword ptr [rdx+0xc0], zmm3
vmovups zmmword ptr [rdx+0x100], zmm4
vmovups zmmword ptr [rdx+0x140], zmm5
vmovups zmmword ptr [rdx+0x180], zmm6
vmovups zmmword ptr [rdx+0x1c0], zmm7
vmovups zmmword ptr [rdx+0x200], zmm8
vmovups zmmword ptr [rdx+0x240], zmm9
vmovups zmmword ptr [rdx+0x280], zmm10
vmovups zmmword ptr [rdx+0x2c0], zmm11
vmovups zmmword ptr [rdx+0x300], zmm12
vmovups zmmword ptr [rdx+0x340], zmm13
vmovups zmmword ptr [rdx+0x380], zmm14
vmovups zmmword ptr [rdx+0x3c0], zmm15
vmovups zmmword ptr [rdx+0x400], zmm16
vmovups zmmword ptr [rdx+0x440], zmm17
vmovups zmmword ptr [rdx+0x480], zmm18
vmovups zmmword ptr [rdx+0x4c0], zmm19
add rdx, 0x500
add rdi, 0x500
cmp r12, 0x9
jl 0x1e
pop r15
pop r14
pop r13
pop r12
pop rbx
ret
```