Extreme Scale FMM-Accelerated Boundary Integral Equation Solver for Wave Scattering: Supplementary Material

MUSTAFA ABDULJABBAR[1,*], MOHAMMED AL FARHAN[1,*], NOHA AL-HARTHI[1], RUI CHEN[1], RIO YOKOTA[2], HAKAN BAGCI[1], AND DAVID KEYES[1]

[1] *King Abdullah University of Science and Technology, Thuwal, Saudi Arabia*

[2] *Tokyo Institute of Technology, Tokyo, Japan*

[*] *Corresponding author: mustafa.abduljabbar@kaust.edu.sa*

November 9, 2018

BEMFMM (https://ecrc.github.io/BEMFMM/) is an extreme-scale Fast Multipole Method (FMM)-accelerated Boundary Element Method (BEM) parallel solver framework. It is a boundary integral equation solver for wave scattering suited for many-core processors, which are expected to be the building blocks of energy-austere exascale systems, and on which algorithmic and architecture-oriented optimizations are essential for achieving worthy performance. It uses the GMRES iterative method and FMM to implement the MatVec kernel. The underlying kernels are highly optimized for both shared- and distributed-memory architectures. The solver framework features optimal architecture-specific and algorithm-aware partitioning, load balancing, and communication reducing mechanisms. To this end, BEMFMM framework provides a highly scalable FMM implementation that can be efficiently applied to the computation of the Helmholtz integral equation kernel. In particular, it deals with addressing the parallel challenges of such application, especially at extreme-scale settings, with emphasis on both shared- and distributed-memory performance optimization and tuning on emerging HPC infrastructures.

1. EXPERIMENTAL SETUP AND WORKLOAD CHARACTERIZATION

This section describes our experimental platforms, the datasets, and the scientific performance engineering methodologies that are used to analyze and present the performance evaluation results.

A. Software Stack and Hardware Configuration

The source code is written in `C++`. We use `PETSc` release version 3.10 built on top of Intel Parallel Studio version 2018 Update 1, which includes Intel `C/C++` compiler, Threading Building Blocks (TBB), OpenMP, Cilk Plus[1], MPI, and Math Kernel Library (MKL). We use `ParMETIS` version 5.0. `PETSc` scalar type is set to complex. `PETSc` is complied with the `C++` compiler, and the `FORTRAN` kernels are set to generic for faster complex number performance. All experiments are performed with the `-O3` compiler optimization flag, and OpenMP affinity is set to scatter via `KMP_AFFINITY=scatter`. The pinning and binding of the thread contexts and the MPI ranks are set to target a specific `quadrant/tile/core` on KNL, and a specific socket/core on CPU. Furthermore, we use `numactl` Linux command to control binding and interleaving of memory channels. Table S1 summarizes the specifications of the Intel x86 architectures considered herein.

Table S1. Hardware specifications.

	KNL	Haswell (HSX)	Skylake (SKL)
Family	x200	E5V3	Scalable
Model	7290	2670	8176
Socket(s)	1	2	2
Cores	72	32	56
GHz	1.50	2.60	2.10
Watts/socket	245	120	165
DDR4 (GB)	192	128	264
Frequency Driver	acpi-cpufreq	acpi-cpufreq	acpi-cpufreq
Max GHz	1.50	2.60	2.10
Governor	conservative	performance	ondemand
Turbo Boost	✓	✓	✓

For the large-scale experiments, we use KAUST's Shaheen XC40, the rank 29 supercomputer according to the TOP500's June 2018 announcement. The system consists of 6,174 compute nodes, each of which is equipped with a dual socket Intel Haswell CPU (see Table S1). The entire system has 197,568 hardware cores, and 786 TB of main memory. (Note that in our experimentations we consider the full scale of Shaheen is 6,144 compute nodes with 196,608 hardware cores, by which we purposely leave 30 compute nodes (i.e., 960 hardware cores) untouched for logistical configurations.) The compute nodes are connected by the Cray Aries interconnect with dragonfly topology, which provides for a maximum of 3 hops for a message between any pair of nodes. Theoretically, Shaheen has a peak double precision floating point

[1]Intel Cilk Plus is being deprecated in the 2018 release of Intel Software Development Tools.

performance of 7.2 PFlop/s.

On Shaheen, We use `PETSc` release version 3.10 built on top of GNU GCC/G++ Compiler version 7.0, GNU OpenMP, and Intel TBB for the GCC Compiler. We do not configure Cilk Plus on Shaheen. We use BLAS and LAPACK open source implementation, and we use Cray `MPICH` implementation that is built on top of GNU compiler. We use `ParMETIS` version 5.0. `PETSc` scalar type is set to complex. `PETSc` is complied with the `C++` compiler, and the `FORTRAN` kernels are set to generic for faster complex number performance.

The settings of the experiments are similar to the aforementioned single node settings with negligible differences in the command line variables and threading tunning instructions, which are are adjusted according to the underlying software stack of Shaheen.

B. Dataset Description

Table S2 describes the specifications of our experimental datasets.

Table S2. Spherical Mesh dataset descriptions.

Mesh	Elements	Nodes	Edges	Number of unknowns (N)
A	156	312	468	936
B	3,156	6,312	9,468	18,936
C	7,274	14,548	21,822	43,644
D	14,338	28,676	43,014	86,028
E	22,370	44,740	67,110	134,220
F	41,258	82,516	123,774	247,548
G	60,204	120,408	180,612	361,224
H	93,590	187,180	280,770	561,540
I	115,454	230,908	346,362	692,724
J	159,288	318,576	477,864	955,728
K	250,514	501,028	751,542	1,503,084
L	314,212	628,424	942,636	1,885,272
M	374,360	748,720	1,123,080	2,246,160
N	1,497,440	2,994,880	4,492,320	8,984,640
O	5,989,760	11,979,520	17,969,280	35,938,560
P	23,959,040	47,918,080	71,877,120	143,754,240
Q	95,836,160	191,672,320	287,508,480	575,016,960
R	383,344,640	766,689,280	1,150,033,920	2,300,067,840

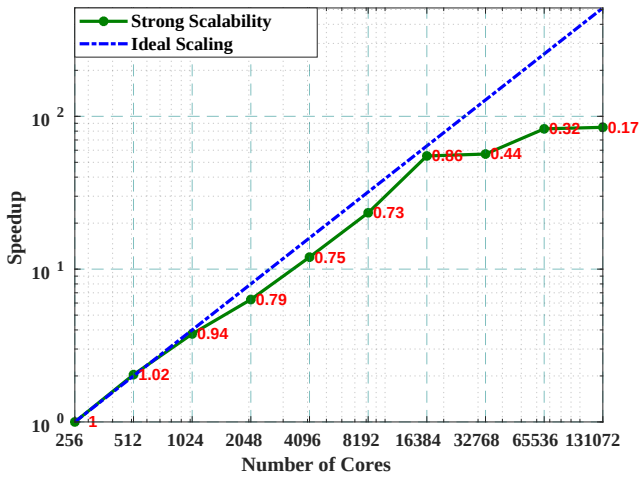
C. Experimental Setup

To report the most accurate performance measurements irrespective of the hardware states and conditions, we apply several state-of-the-practice scientific performance engineering methodologies, which overcome any possible hardware-

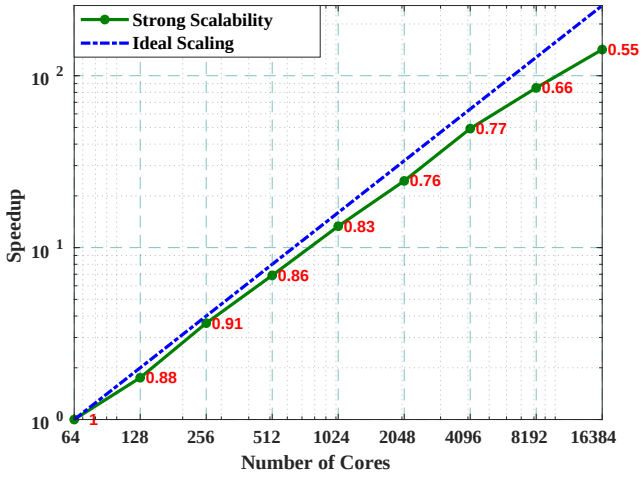
oriented performance variations.

The reported runtime results are summarized using the arithmetic mean across multiple independent runs, which form the sample space. The reported floating point rates, on the other hand, are summarized using the harmonic mean [1, 2]. Unless otherwise reported, we average approximately 50 runs for every experiment, except for the large-scale results, in which we are constrained by the available core hours. Thus, we reduce the size of the sample space for every run based on the available core hours (i.e., we roughly average between 5 to 10 runs for every experiments, based upon the problem size and the wall-clock time of a specific run). In addition, an error bar is drawn to show the +/- standard deviation of the mean for each experimental sample.

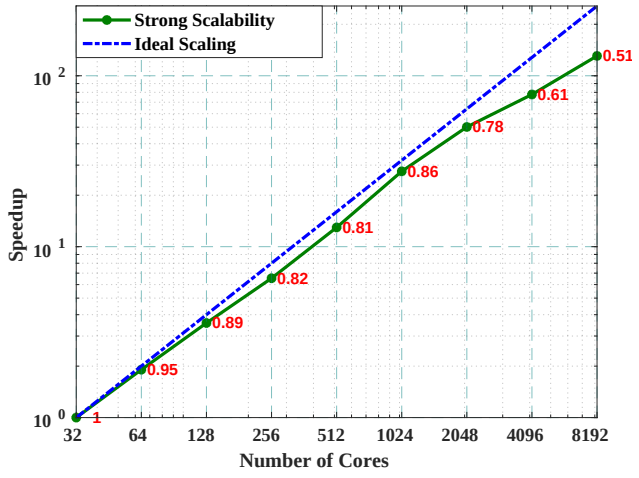
2. STRONG SCALABILITY STUDY ON SMALL DATASETS



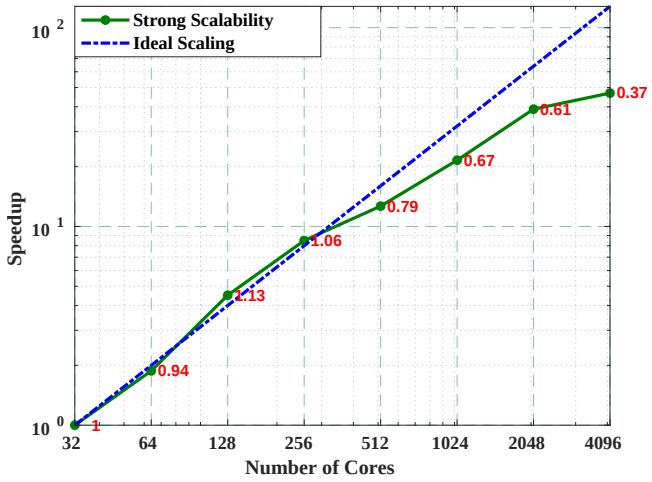
(a) 35,938,560 DoF



(b) 8,984,640 DoF



(c) 2,246,160 DoF



(d) 561,540 DoF

Fig. S1. Weak and strong scalability results on Shaheen from 1 to 4,096 compute nodes. Performance is normalized by the FMM time per iteration and the total number of GMRES iterations. Numbers along the graph lines (red) indicate efficiency with respect to the ideal speedup (efficiency baseline is the smallest core count).

3. DUFFY TRANSFORMATION: TREATING $1/R$ SELF-SINGULARITY

Duffy's method is a technique that employs a change of integration variable, where each transformation scheme results in different forms of Jacobian that cancels the singularity in the kernel [3]. This method is mainly applied to $1/R$ self-singularity cases. Figure S2 gives a bird's-eye view of the variable transformation undertaken by Duffy's method, by which the singularity is canceled through multi-transformations, and then the integral can be evaluated using standard numerical techniques such as Gaussian integration rule.

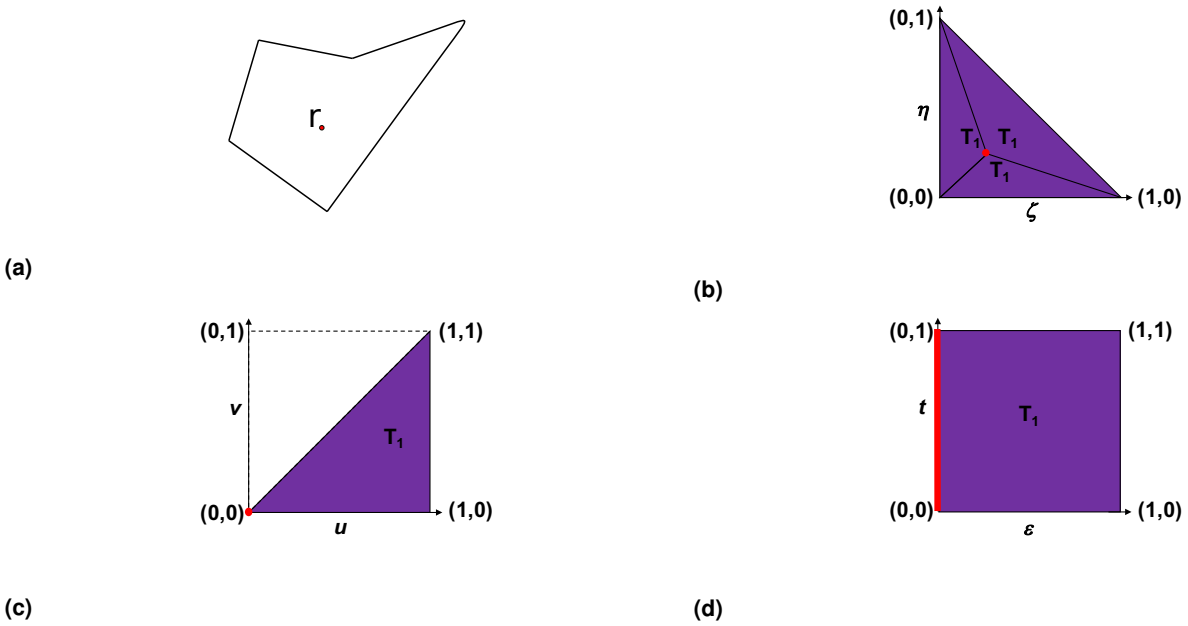


Fig. S2. Variable transformation by Duffy's method.

Following summarizes the multi-transformations done by the Duffy's method:

1. Figure S2a shows the integration to evaluate over curvilinear source patch, where the field point lies inside the patch.
2. Figure S2b presents the integration mapping to the reference triangle.
3. The reference triangle is partitioned into three sub-triangles that share the singular point.
4. The integration on the reference triangle can be obtained by the summation of the integrations on the three sub-triangles: T1, T2 and T3.
5. Figure S2c illustrates the mapping into a (u, v) space of Figure S2b sub-triangle, which in turn maps the singular point to the origin point.
6. Figure S2d exhibits the (u, v) space mapping into a square (ϵ, t) space, by which it elapses the singularity at one point to one edge of the unitary patch.

Adjacent Singularity: The oscillatory behavior that inhibits convergence of the Green's function occurs when two points are close to each other (i.e., adjacent points) but each one resides in a different patch. For a particular case,

the so-called *P-refinement* is exploited to tackle such oscillatory behavior. *P-refinement* is a well-known technique that simply adds more integration points within the underlying triangular elements, through which we increase the elements' polynomial degrees [4]. This allows convergence of the Green's function at a faster rate.

4. DATA-LEVEL PARALLELISM

Contemporary processing hardware is equipped with Instruction Set Architecture (ISA) that supports Single Instruction, Multiple Data (SIMD) operations on many vectorized data items. For instance, Intel Xeon Skylake architecture implements two 512-bit Vector Processing Units (VPUs) per core, by which a single arithmetic instruction can be performed on a large subset of independent, distinct data items. In the context of our highly optimized FMM Helmholtz kernels, we undertake two different vectorization approaches: 1) we handwrite the vector code for the key kernels using AVX-512 intrinsics, and 2) we further optimize and fine-tune the kernels to aid the Intel compiler to automatically generate efficient vector codes.

Validating the Compiler's Loop Choice: Treating singularity in the innermost loop of the 3D Helmholtz kernel depicted by Listing 1 through iterating over the high degree Gauss quadrature points within the Particle-to-Particle (P2P) and Source-to-Target (S2T) routines of FMM involves, in principle, complicated nested `for loops` that involve many conditional statements. When we analyze the Intel compiler's report of vectorization generated by the Intel Advisor assistance tool, we find out that the compiler tends to vectorize the innermost loops by default. However, this does not result in an additional advantage out of the vectorization, since the spatial and temporal locality of references are well-preserved when the outer loops are vectorized and strided [5–7]. One way to vectorize such kernels is through populating the scalar loop's data using several vector broadcast instructions, as opposed to fetching a unit stride from a cache line via `vmov` instruction. Thus, using `vbroadcast` instructions imposes lower latency and reciprocal throughput [8]. Furthermore, lowest latency and highest spatial and temporal locality of reference are achieved with outer loop vectorization, especially in balanced (equal-sized) chucks inside the nested `for loops`. In order to construct such vector code, either the compiler needs to be instructed via `#pragma simd`, or more aggressively, writing the vector code manually through utilizing intrinsics. Indeed, both approaches require certain loop optimization techniques (e.g., transformation and unrolling) [1, 9]. Relying on the compiler to auto-vectorize the code, whenever it is possible, is definitely the right approach and is highly recommended to guarantee portability and resilience. Nonetheless, in many cases the compiler could fail to extract the correct or efficient vector code due to assumed data dependencies imposed by the data structures.

Writing SIMDizable Code: Since some arithmetics are known to be expensive in terms of latency, which could squander many CPU cycles, modern compilers are designed to avoid such arithmetics as much as possible. For instance, square root and division operations are very often replaced by their reciprocal counterpart, whenever the code is compiled with certain optimization flags. Nevertheless, sometimes the compiler's auto-generated vector codes is suboptimal, which primarily depends on the scalar source code. Hence, writing intrinsics seems to be inevitable in such cases. For example, consider $1/sqrt(R)$ of line 19 in Listing 1, the corresponding assembly code of the Intel compiler comprises of 2 `vmovups`, 3 `vmulps`, 1 `vrsqrt14ps`, 1 `vfmsub213ps`. This is fairly a reasonable approach that the compiler adopts

```
for ( ; i<ni; ++i) {
  vi_r = real(Bi.SRC); vi_i = imag(Bi.SRC);
  for (j=0; j<nj; ++j) {
    dX = xi-xj;
    R2 = norm(dX);
  \\relay self-sigularity to PETSc callback
    if(Bi.PATCH!=Bj.PATCH && R2!=0) {
      real_t R=sqrt(R2);
      if(R<=near_patch_distance) {
        for (k=0; k<gauss_quad_points; ++k) {
        \\ near patch singularity treatment
        } } else {
        vj_r = real(Bj.SRC); vj_i = imag(Bj.SRC);
        src2_r = vi_r*vj_r-vi_i*vj_i;
        src2_i = vi_r*vj_i+vi_i*vj_r;
        invR = 1.0/sqrt(R);
        eikr = 1.0/exp(wave_i*R);
        eikr *= invR;
        eikr_r = cos(wave_r*R)*eikr;
        eikr_i = sin(wave_r*R)*eikr;
        pot_r += src2_r*eikr_r-src2_i*eikr_i;
        pot_i += src2_r*eikr_i+src2_i*eikr_r;
      } } }
  Bi.TRG += complex(pot_r, pot_i);
}
```

Listing 1. Trimmed down version of the complex-valued 3D Helmholtz P2P kernel.

to build portable, efficient vector code. However, one can write a more efficient code, which uses only `vmovups` and `vrsqrt14ps`, via an explicit call to `_mm512_rsqrt14_ps(r)` intrinsic. On the other hand, a smarter way can achieve both (i.e., efficient vector code generated by the compiler while avoiding writing explicit SIMD code) through breaking down the $eikr = (\sqrt{R} \times e^{ikR})^{-1}$ operation to lines 19, 20, and 21 of Listing 1. Thereby, the compiler automatically understands this transformation, and would extract the most cost-effective, well-optimized vector code.

Optimizing Memory Access: It is well-understood that memory bandwidth is a critical obstacle that limits the performance of modern HPC architectures. As a consequence, one must carefully inspect how cache lines or memory words are fetched into the vector units, especially since most modern x86 architectures are mounted on dual-socket NUMA nodes, in which data might physically reside on different address spaces [1]. Therefore, we develop the FMM core kernels to allocate and reference the particles and tree cells data structures in the form of Array-of-Structs (AoS). In addition, AoS enhances the locality of references for interacting particles after they are sorted and indexed based on their Morton order. Cells maintain both indexes and counts of the encapsulated set of particles (see Listing 2). This is somehow a simple and compact version of a *hash map* associative array abstract data type to map keys to values (i.e., index and count). Hence, strided memory access via AVX-512 intrinsics can be utilized to efficiently reference the SRC data structure of line 2 of Listing 1. However, having to carry out such low-level manipulation with intrinsics might not be attainable except with a great deal of coding effort, since it requires manipulation of memory addresses using `shuffle`, `permute`, `gather` and `scatter` instructions, which results in non-portable, error-prone, compiler-specific code.

```
struct particle_t {
  num SRC;
  num COORD[3];
} __attribute__((aligned (64)));

struct cell_t {
  particle_t* b_ptr;
  size_t b_count;
} __attribute__((aligned (64)));

particle_t* particles;
cell_t*     cells;
```

Listing 2. Summary of underlying FMM's data structures.

Thus, we code the low-level kernels in such a way that the compiler can extract the most optimal vector code without the need to explicitly use handwritten intrinsics.

5. CONVERGENCE EFFECTS OF THE SINGULARITY TREATMENTS

Figure S3 shows the convergence behavior of the solver where we calculate the far scattered fields interactions by a sphere of radius $1m$ using different singularity treatment modes. The Y-axis shows the number of iterations required for GMRES convergence, whereas the X-axis represents the GMRES relative residual norm. The self-singularity treatment creates higher order Gauss quadrature points around the sources and targets falling exactly on the diagonal, or having $[R < \epsilon]$ from geometrical perspective. Such points are ignored in typical FMM implementations. Near-singularity scheme treats points that fall within $[R < near]$ radius (*near* is a code parameter). It can be clearly depicted in Figure S3 that when singularity is ignored, we exhibit a lower convergence rate that is driven and dominated by the singularity of the Green's function. However, when treating only the true singularity, a remarkably improved convergence rate is observed. Finally, when considering the self- and near-singularity treatment schemes, the relative 2-norm residual accuracy reached 1.0e-4 within just less than 50 iterations. This accuracy corresponds to 1.0e-3 to 1.0e-4 error with respect to the analytical solution, and it cannot be further improved even with less relative tolerance of the underlying iterative solution. Hence, our GMRES solver is configured to exit at 1.0e-4 relative 2-norm residual accuracy.

REFERENCES

1. M. A. A. Farhan and D. E. Keyes, "Optimizations of unstructured aerodynamics computations for many-core architectures," IEEE Transactions on Parallel Distributed Syst. **29**, 2317–2332 (2018).
2. T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*, (ACM, 2015), pp. 73:1–73:12.
3. M. G. Duffy, "Quadrature over a pyramid or cube of integrands with a singularity at a vertex," SIAM J. on Numer. Analysis **19**, 1260–1262 (1982).

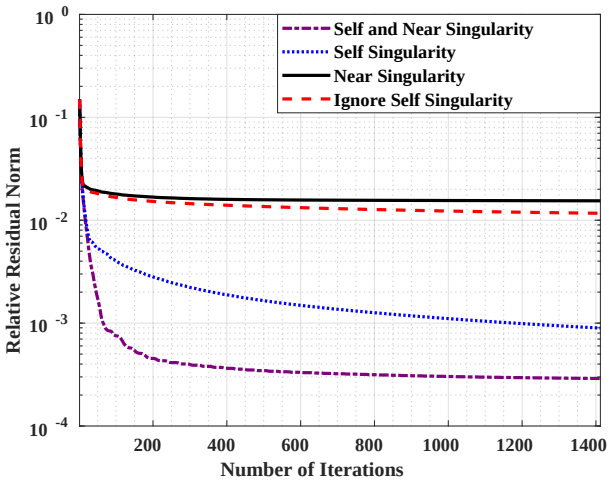


Fig. S3. Convergence effects of self- and near-singularity treatments.

4. I. Babuska, B. Szabo, and I. Katz, "The p-version of the finite element method," SIAM J. on Numer. Analysis **18**, 515–545 (1981).
5. N. Dorit and A. Zaks, "Outer-loop vectorization - revisited for short SIMD architectures," in *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, (2008).
6. M. Abduljabbar, M. Al Farhan, R. Yokota, and D. Keyes, *Performance Evaluation of Computation and Communication Kernels of the Fast Multipole Method on Intel Manycore Architecture* (Springer International Publishing, Cham, 2017), vol. 10417 of *Lecture Notes in Computer Science*, pp. 553–564.
7. M. Abduljabbar and R. Yokota, "N-body methods," in *High Performance Parallelism Pearls*, (Morgan Kaufmann - Elsevier, Burlington MA, 2015), chap. 10.
8. "Instruction tables," Agner Fog Res. p. 88 (2017).
9. H. Takizawa, T. Reimann, K. Komatsu, T. Soga, R. Egawa, A. Musa, and H. Kobayashi, "Vectorization-aware loop optimization with user-defined code transformations," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, (2017), pp. 685–692.