Embracing a New Era of Highly Efficient and Productive Quantum Monte Carlo Simulations

Amrita Mathuriya
Intel Corporation
amrita.mathuriya@intel.com

Ye Luo
Argonne National Laboratory
yeluo@anl.gov

Raymond C. Clay III
Sandia National Laboratories
rclay@sandia.gov

Anouar Benali
Argonne National Laboratory
benali@anl.gov

Luke Shulenburger
Sandia National Laboratories
lshulen@sandia.gov

Jeongnim Kim
Intel Corporation
jeongnim.kim@intel.com

ABSTRACT

QMCPACK has enabled cutting-edge materials research on supercomputers for over a decade. It scales nearly ideally but has low single-node efficiency due to the physics-based abstractions using array-of-structures objects, causing inefficient vectorization. We present a systematic approach to transform QMCPACK to better exploit the new hardware features of modern CPUs in portable and maintainable ways. We develop miniapps for fast prototyping and optimizations. We implement new containers in structure-of-arrays data layout to facilitate vectorizations by the compilers. Further speedup and smaller memory-footprints are obtained by computing data on the fly with the vectorized routines and expanding single-precision use. All these are seamlessly incorporated in production QMCPACK. We demonstrate upto 4.5x speedups on recent Intel® processors and IBM Blue Gene/Q for representative workloads. Energy consumption is reduced significantly commensurate to the speedup factor. Memory-footprints are reduced by up-to 3.8x, opening the possibility to solve much larger problems of future.

KEYWORDS

QMC, vectorization, optimizations, portability, CPUs

ACM Reference Format:
Amrita Mathuriya, Ye Luo, Raymond C. Clay III, Anouar Benali, Luke Shulenburger, and Jeongnim Kim. 2017. Embracing a New Era of Highly Efficient and Productive Quantum Monte Carlo Simulations. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
https://doi.org/10.1145/3126908.3126952

1 INTRODUCTION

Large-scale parallel computing resources have enabled numerous science discoveries and grand-challenge simulations since the early 1990s. Productive utilization of high-performance computing (HPC) resources demands algorithms and implementations that are both highly efficient and scalable. The gap between the peak and sustained performance that a typical HPC application can achieve has been steadily growing. The news article "4 applications sustain 1 petaflop on Blue Waters" in 2013 [1] manifests the challenges the developers are facing to exploit the powerful systems at scale.

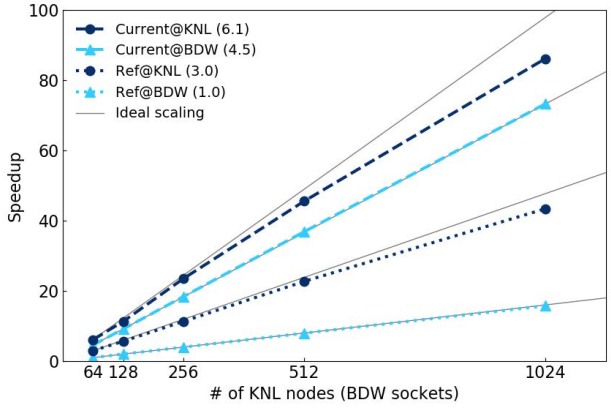

Figure 1: Strong scaling of NiO-64 benchmark on Trinity at LANL (KNL) and Serrano at SNL (BDW) systems. The performance is normalized by a reference throughput using 64 BDW sockets. Slopes of the ideal-scaling lines are provided in parentheses.

Multiple factors are responsible for the growing performance gap. The increasing complexity of HPC applications, a fast evolving hardware landscape, and a wide range of programming models offered to the developers — all play roles in the decreasing productivity of extremely capable HPC systems. Lately, much of the increase in computing power of the processors comes from increasing opportunities for parallelism on a node through many cores, multiple hardware threads and wide SIMD units. Without fully exploiting these parallelisms and unique hardware features, such as high-bandwidth memory and cache subsystems, applications leave a lot of potential performance gain on the table. However, any change of a production-level HPC application to adopt and adapt to new HPC infrastructure is a formidable task even for a team of highly experienced developers. This work explores portable and maintainable methods to transform

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126952

QMCPACK [2], which has similar compute and design characteristics to many HPC applications, to achieve significantly more efficient single-node performance.

Quantum Monte Carlo (QMC) is a highly accurate, but computationally demanding method. It consumes a significant fraction of US-DOE resources every year, leveraging highly scalable algorithms and implementations. Typical QMC calculations use 1000s of nodes at a time on the leadership facilities. QMCPACK implements hybrid parallelism with OpenMP and MPI [3] and has close to ideal parallel efficiency as shown in Fig. 1. The figure shows strong scaling of NiO-64 benchmark on 2nd generation Intel® Xeon Phi™ processor (KNL) and Intel® Xeon® E5v4 processor (BDW). However, on-node efficiency is low and it achieves below 10% of the peak performance even on Blue Waters [1]. Compounding this problem, it does not utilize SIMD parallelism to the fullest extent except for special kernels using platform-dependent intrinsics, e. g., QPX intrinsics [4] on IBM BG/Q, or SSE/SSE2 intrinsics on x86. As shown in Fig. 1, our work on QMCPACK increases on-node efficiency by 2-4.5x, which translates directly to a multi-node speedup of the same factor, with nearly ideal scaling. As an added benefit, this increase in compute efficiency impacts not only scientific productivity, but also results in similar improvement in energy efficiency.

This work presents a systematic approach to transform QMCPACK. We use representative workloads of various problem sizes and computational characteristics on multiple platforms to develop a set of miniapps to optimize the most computationally expensive components of the application. The use of miniapps facilitates exploration of a large design space and algorithms and fast prototyping of new methods while maintaining realistic code usage. The full integration of the new solutions is then staged to evaluate the performance impact of each step, minimize the changes in the high-level QMC drivers and validate the correctness of the implementations. We analyze the performance evolution throughout the optimization processes and iteratively improve both miniapps and the full application.

Based on the extensive performance analysis of the current workloads including those used in this work, we set two main targets to increase the performance: i) improve SIMD efficiency and ii) reduce memory footprint. We aim to develop portable and maintainable solutions to increase the productivity of the QMC experts who use QMCPACK to develop new electronic structure theories, numerical techniques and parallel algorithms. Hence, the code transformations are constrained to use C++11 and OpenMP 4 standards, consistent with the existing physics abstractions and the thread-level parallelization in QMCPACK. No platform-specific optimizations are employed for this work. However, the infrastructure — miniapps, classes/interfaces *etc* — is devised to be extensible. Specialization for a specific hardware can be added for further improvement.

We demonstrate the performance impact using four representative workloads. Our work speeds up QMCPACK simulations by 2-4.5x on KNL and BDW clusters of up to 1024 MPI

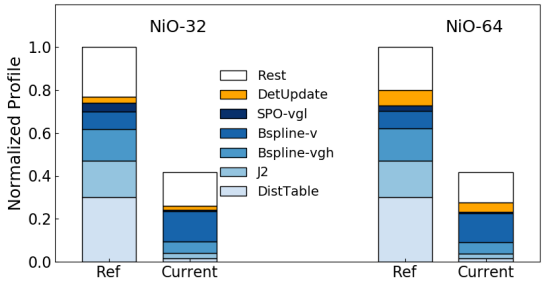

Figure 2: Normalized hot-spot profiles on KNL. Current version profiles accomodate the speedup wrt. Ref version for the corresponding benchmark.

tasks as Fig. 1 shows. The energy usage reduction in proportion to the speedup factor on KNL system is achieved by the optimizations of this work. The memory usage is reduced to fit in KNL's 16GB MCDRAM memory for a large problem with 784 electrons. Our work leads to more productive QMC simulations by enabling users to solve larger problems quicker.

1.1 Summary of work and contributions

A detailed analysis of the latest release reveals low SIMD efficiency in key compute kernels. The top hot-spots of the reference profiles in Fig. 2 are DistTable, J2 and Bspline which use array-of-structures (AoS) data types to represent 3D physics of the electrons, such as the positions of N electrons in `R[N][3]`. This abstraction is the foundation for all high-level algorithms in QMCPACK. However, highly productive abstractions for science can incur a high abstraction penalty as demonstrated by numerous studies [5, 6]. For this reason, we introduce complementary objects of structure-of-arrays (SoA) types for all the compute expensive kernels. For example, `Rsoa[3][N]` for `R` is added to enable efficient vectorization and increase bandwidth utilization.

Also, the memory footprint of QMCPACK grows as $\mathcal{O}(N^2)$ with the number of electrons and can quickly become challenging. Fully utilizing a KNL system requires large number of threads and walkers (samples), making the footprints larger, compared to regular Intel® Xeon® systems. We solve the problem with i) mixed-precision (MP) and ii) compute-on-the-fly algorithms. By expanding single precision use in the key data structures and methods, we reduce memory use and bandwidth demands as well as making the computations faster. Once the key computational kernels become faster by exploiting efficient vectorization, it becomes faster to compute elements when they are used than to store and retrieve them. These changes result in much decreased run time and better memory usage.

Transforming a big application of millions of lines of code and 1000s of files to adopt new data layouts and mixed-precision algorithms is a big and complicated task and must be carried out carefully to improve both application performance and science productivity. The unique properties of

QMC algorithms and the object-oriented and generic framework of QMCPACK are exploited to transform the full application. New SoA objects are added to improve the SIMD efficiency in the critical routines and the existing abstractions and AoS objects are reused. The miniapps facilitate fast prototyping and evaluations and minimize the risk of the global transformations until they are proven to be effective in realistic QMC simulations.

In summary, **Contributions** of this publication are following:

- Created miniapps representing compute and data access patterns of QMC simulations and used them to integrate the new developments to the full production QMCPACK.
- Implemented SoA data types, facilitating efficient vectorization of all the compute intensive kernels using C++11 and OpenMP standards.
- Developed forward update and compute-on-the-fly algorithms, enabling further speedup and memory footprint reduction.
- Expanded single-precision use in CPU code for memory reduction and speed and improved the accuracy of the mixed-precision methods for both CPU and GPU ports.

2 RELATED WORK

Microkernels or miniapps have been widely used for HPC procurements or acceptance testing. For instance, the CORAL microkernel benchmarks are code snippets extracted from HPC applications. These are intended to address certain capabilities of a system, such as NEKbonemk used for SIMD compiler challenge [7]. The miniapps of this work are intended to spur QMC development, going beyond the traditional roles of microkernels. They reproduce the computational patterns, memory usage, data access and thread-level parallelism of the full code as realistically as possible. Performance changes in these miniapps are reliable predictors of the performance of real QMC simulations. We use them to narrow the solution space for the optimization and parallelization of QMCPACK.

Our previous work [8] showed performance improvement in 3D B-spline routines using a SoA data type. In this work, we implement the SoA data types in the full QMCPACK code for the top kernels. Also, we use generic C++ containers to port the optimizations instead of using plain old data types.

`VectorSoaContainer<T,D>` (VSC) adopts the concepts of SIMD Data Layout Templates (SDLT) library introduced in Intel® C++ Compiler 17.0 [9, 10]. VSC is a generic SoA container of `C[D][N]` for D dimensional particle simulations, providing access operators and methods. Current SDLT only supports "plain old data" objects and adopting it in QMCPACK would require large-scale refactoring while losing the generality it aims to maintain. Therefore, we introduce VSC to express the high-level QMC algorithms as before, while hiding the implementation details — memory allocation and layout. Very limited changes are made at the physics abstraction level and they are mostly to incorporate new algorithms.

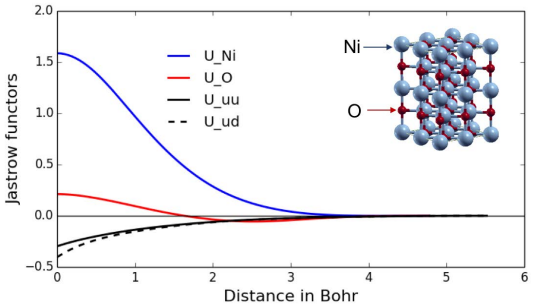

Figure 3: Jastrow functors of Ni and O ions and up and down electron spins for a 32-atom supercell of NiO.

Single precision has been extensively used in QMCPACK's GPU port [11] resulting in significant speedups and memory savings. Single precision was later introduced to the CPU version to compute the 3D B-spline SPOs (single-particle orbitals) [12]. This work expands the use of single precision to the entire QMC calculations. To preserve numerical accuracy for both CPU and GPU ports, the quantities per walker and for the ensemble are computed in double precision and new states are periodically computed from scratch [13].

QMCPACK makes extensive use of object-oriented and generic programming and design patterns [14] for reusability and extensibility [3]. Computational efficiency is achieved through inlined specializations of C++ template and by using SIMD intrinsics for core kernels [11, 12]. This work eliminates the platform-dependent optimization and leverages optimizing C++ compilers and OpenMP standards to achieve greater efficiency on modern CPUs. Many features in C++11 [15] are used to make the code compact, efficient and maintainable.

3 QMC ALGORITHMS

In quantum mechanics, all physically observable quantities for a system containing N particles can be computed from the $3N$-dimensional *wave function*, $\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)$ [16]. For any trial *wave function*, $\Psi_T(\mathbf{R})$, we can compute an energy as the expectation value of the many-body Hamiltonian, $\hat{H}$,

$$E_T = \frac{\int d^{3N}\mathbf{R}\ \Psi_T^*(\mathbf{R})\hat{H}\Psi_T(\mathbf{R})}{\int d^{3N}\mathbf{R}\ |\Psi_T(\mathbf{R})|^2}, \tag{1}$$

where $\mathbf{R}$ is a $3N$-dimensional vector representing the positions of the N particles. The direct evaluation of many-dimensional integrals of Eq. (1) by stochastic sampling enables us to employ highly accurate variational wave functions which can capture crucial many-body effects in an efficient manner. The Slater-Jastrow trial wave function used in this work is

$$\Psi_T = \exp(J)D^u(\{\phi\})D^d(\{\phi\}), \tag{2}$$

with $N = N^u + N^d$ for the up and down spins. For the rest of the paper, we assume $N^u = N^d = N/2$.

The Jastrow factor J describes the dynamic correlation and is factorized into one-body, two-body and high-order

correlation functions as

$$J = \sum_{I}^{N_{\rm ion}} \sum_{i}^{N} U_I(|\mathbf{r}_I - \mathbf{r}_i|) + \sum_{j \neq i}^{N} U_2(|\mathbf{r}_i - \mathbf{r}_j|) + \cdots \quad (3)$$

Figure 3 shows distinct Jastrow functors optimized for a 32-atom supercell of NiO. The one-dimensional cubic B-spline is extensively used in QMCPACK because of its generality and computational efficiency [17].

The Slater determinant captures the static correlation and ensures the antisymmetric property of a Fermionic wave function upon exchange of a pair of electrons as $D = \det|\mathbf{A}|$ and $A(i,j) = \phi_i(\mathbf{r}_j)$. Here, $\{\phi\}$ denotes a set of SPOs, often taken to be the solution of a mean-field method such as density functional theory or the Hartree-Fock approximation.

The diffusion Monte Carlo algorithm (DMC) shown in Alg. 1 is the most time-consuming stage of a QMC exploration of a system. We can define the efficiency of a DMC calculation as $\kappa = 1/(\sigma^2 \tau_{\rm corr} T_{\rm MC})$, where σ is the variance for the optimized Ψ_T. Increasing computational and parallel efficiency impacts the DMC efficiency by reducing the total MC time $T_{\rm MC}$ to reach the target statistical error. The auto-correlation time $\tau_{\rm corr}$ [18] reflects the quality of Ψ_T and the MC algorithms. The ensemble size, the average number of walkers, is important to reduce systematic errors due to the finite time step and population.

Algorithm 1 Pseudocode for diffusion Monte Carlo.

```
1: for MC generation = 1 ··· M do
2:   for walker = 1 ··· N_w do
3:     let R = {r_1 ... r_N}
4:     for particle k = 1 ··· N do
5:       set r'_k ← r_k + ∇_k Ψ_T(R) + δ
6:       let R' = {r_1 ... r'_k ... r_N}
7:       ratio ρ = Ψ_T(R')/Ψ_T(R)
8:       derivatives ∇_k Ψ_T(R'), ∇²_k Ψ_T(R')
9:       Accept r_k ← r'_k or reject
10:    end for{particle}
11:    local energy E_L = Ĥ Ψ_T(R)/Ψ_T(R)
12:  end for{walker}
13:  reweight and branch walkers
14:  update E_T and load balance
15: end for{MC generation}
```

A typical DMC implementation employs a particle-by-particle (PbyP) update for the *drift-and-diffusion stage* (L4-L10) to increase the MC efficiency. Only one particle is moved at a time in this algorithm. Once a new configuration is sampled, the physical quantities, such as the local energy E_L, are measured for the fixed electron positions. The rest consists of computing the trial energy E_T, taking statistics and load balancing of the fluctuating population.

The Slater-Jastrow form used for the trial wavefunction, Ψ_T, Eq. (2-3) is physically motivated but also has many computational advantages for the PbyP update. Take the step of moving the k-th electron from $\mathbf{r}_k$ to $\mathbf{r}'_k$. The computation of the ratio becomes

$$\frac{\Psi_T(\mathbf{r}_1 \cdots \mathbf{r}'_k \cdots \mathbf{r}_N)}{\Psi_T(\mathbf{r}_1 \cdots \mathbf{r}_k \cdots \mathbf{r}_N)} = \exp^{\Delta J_1} \exp^{\Delta J_2} \frac{\det|\mathbf{A}'|}{\det|\mathbf{A}|}, \quad (4)$$

where

$$\begin{aligned} \Delta J_1 &= \textstyle\sum_I^{N_{\rm ion}} U_I(|\mathbf{r}_I - \mathbf{r}'_k|) - \sum_I^{N_{\rm ion}} U_I(|\mathbf{r}_I - \mathbf{r}_k|), \\ \Delta J_2 &= \textstyle\sum_{i\neq k}^{N} U_2(|\mathbf{r}_i - \mathbf{r}'_k|) - \sum_{i\neq k}^{N} U_2(|\mathbf{r}_i - \mathbf{r}_k|). \end{aligned} \quad (5)$$

The determinant ratio is a dot product of the k-th row of $\mathbf{A}^{-1}$ and $v(\phi_1(\mathbf{r}'_k), \cdots, \phi_{N/2}(\mathbf{r}'_k))$ using

$$\det(\mathbf{A} + ue'_k) = (1 + e'_k \mathbf{A}^{-1} u) \det(\mathbf{A}). \quad (6)$$

The derivatives for the quantum forces on the electron are evaluated using the same matrix determinant lemma [19, 20]. When the proposed $\mathbf{r}'_k$ is accepted, $\mathbf{A}^{-1}$ is updated using Sherman-Morrison formula. Other internal states, such as the distance tables between the electrons and ions for Jastrow computations are updated to proceed to the next particle move.

Once a new configuration is obtained, E_L is computed as

$$E_L = -\frac{\nabla^2 \Psi_T(\mathbf{R})}{2\Psi_T(\mathbf{R})} + \sum_{i<j} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} + \sum_I \frac{\hat{V}_{\rm NL} \Psi_T(\mathbf{R})}{\Psi_T(\mathbf{R})}. \quad (7)$$

The non-local pseudopotential operator $\hat{V}_{\rm NL}$ is handled by approximating an angular integral by a quadrature on a spherical shell surrounding each ion [19]. This requires ratio evaluations of the electrons within a cutoff radius of an ion using Eq. (4).

4 SCIENCE GOALS

Whereas diffusion Monte Carlo has in the past been applied to calculate the properties of idealized highly crystalline materials with high accuracy, the additional computing power that will be brought to bear as supercomputing pushes past the petascale and into the exascale will bring with it the possibility of treating the complexity of realistic materials. If properly harnessed, this could enable new kinds of scientific problems to be addressed. For example, it could be possible to study the aging of photovoltaic materials exposed to the environment rather than just their performance in a laboratory.

In order to meet these goals, however, a code will have to cope with several changing features of the exascale landscape. Firstly, increasing parallelization is arriving often in the form of ever wider vector units instead of increasing numbers of computing cores and secondly, the memory per core is not necessarily increasing at a pace that will satisfy quantum Monte Carlo's $O(N^2)$ memory footprint. To successfully deal with these hurdles, a code will have to increase vectorization while being as conservative as possible with memory utilization.

4.1 Benchmark problems

In this work we will consider four different benchmark systems in order to demonstrate how these algorithmic improvements in QMCPACK have addressed these challenges. The first is a classic throughput based benchmark which was included

Table 1: Workloads used in this work and their key properties.

	Graphite	Be-64	NiO-32	NiO-64
N	256	256	384	768
N_{ion}	64	64	32	64
N_{ion}/uint cell	4	2	4	4
# of uint cells	16	32	8	16
Ion types (Z^*)	C (4)	Be (4)	Ni(18), O(6)	
# of unique SPOs	80	81	144	240
FFT grid	28x28x80	84x84x144	80x80x80	
B-spline (GB)	0.1	1.4	1.3	2.1

in the assessment criteria for the CORAL machines[7]. That benchmark requires calculating the energy of a crystalline domain of graphite, the precursor material for generating graphene. The second benchmark requires the calculation of the properties of beryllium. This system was chosen because it has a similar number of electrons (and hence computational scaling) as the graphite benchmark, but as it is a lighter element, it can be performed without the use of pseudopotentials. The pseudopotentials are a crucial algorithmic consideration necessary for treating heavier elements, but from a computational point of view, their use stresses parts of the algorithm that are not expected to be as important as the size of the problem increases.

The final two benchmarks are closely related. They perform calculations on crystals of NiO, an electronically strongly correlated material that is difficult to treat for many methods. These benchmarks involve calculations on 32 and 64 atom supercells of NiO and provide the most direct assessment of the sort of calculations that are expected to be scientifically important in the near future. Table 1 summarizes key features of these four benchmarks, including the numbers of electrons in each and the number of single particle orbitals required to calculate the trial wavefunction for each one. As the number of electrons is the single most important factor affecting the performance profile, in most of the discussion that follows we will focus on only the NiO 32 and 64 atom benchmarks. These cases involve pseudopotentials, as will most common QMCPACK workloads, and they allow the effects of the changing electron count to be addressed in a direct manner. However, we will refer to the entire set where appropriate to demonstrate the universality of the algorithm across problem types.

5 SYSTEM DETAILS

We used two different shared memory multi/many-core processors to capture performance evolution at each major step: i) dual socket Intel® Xeon® E5v4 CPU (BDW) and ii) second generation Intel® Xeon Phi™ processor 7250P (KNL). We also use IBM Blue Gene/Q (BG/Q) processor for demonstrating portability of our performance improvements. Two types of systems are used for multi-node scaling and performance analysis: i) Trinity at Los Alamos National Laboratory with KNL processors and Cray Aries Dragonfly interconnect; ii) Serrano cluster at Sandia National Laboratories with dual-socket BDW processors and Intel® Omni-Path interconnect.

Two different BDW SKUs are used: i) 20-core single socket E5-2698 v4 CPU for the single-node performance analysis, and ii) 18-core dual socket E5-2695 v4 for multi-node runs on Serrano cluster. KNL processor is used in Quad cluster mode and wherever possible, KNL-MCDRAM is used in flat mode. For a few runs, memory footprints exceed 16GB MCDRAM capacity; those are done in MCDRAM-cache mode on KNL. We use 64 out of 68 cores on the KNL machine, leaving a few cores out to do OS related tasks. Performance comparisons are done between a KNL node and single-socket of a dual-socket BDW node considering their power budgets and NUMA characteristics.

We use tools[1] from Intel® Parallel Studio XE 2017 [21] on Intel® platforms. BDW and KNL use architecture specific compiler options [22]. For advanced hot-spot profiling, Intel® VTune™ Amplifier 2017 (VTune)[23] was utilized. Roofline performance analysis was done with an engineering version based on Intel® Advisor 2017 update 2 [24]. On BG/Q, we used Clang compiler version 4.0.0 (bgclang r284961-stable) [25].

6 REFERENCE QMCPACK

The baseline of this work uses the latest public release of QMCPACK 3.0.0 [2] with the mixed precision feature turned off. This section describes the reference QMCPACK implementation and presents an analysis of its performance.

6.1 Baseline with AoS data types

Figure 4 presents a simplified QMC code, containing a driver method `psuedo_qmc` and core abstractions for D-dimensional particle simulations. It is constructed to mimic the structure of QMCPACK 3.0.0. The threading is implemented with OpenMP. `ParticleSet` and `TrialWaveFunction`, the main compute objects, are created per thread as denoted by `E_th` and `Psi_th`. Here `nw` is a dynamic variable during a DMC run which represents number of walkers. It is updated during the reweight and branch walkers step (L13 in Alg. 1). A generic `Vector<G>` is used to represent any attribute such as positions. The most basic and important attribute `R` encapsulates the positions of N particles in an AoS type, `Vector<TinyVector<T,D>>`.

A `Walker` object is a simple container to manage the positions, physical quantities such as E_L, weight, age etc and an anonymous `Buffer` to store internal state for fast PbyP updates. The exact form and the composition of Ψ_T is only

[1] Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

```
1  //fixed D-dimensional vector for each particle
2  template<typename T, unsigned D>
3  class TinyVector { T X[D]; };
4
5  //generic 1D container
6  template<typename G> class Vector{
7    std::vector<G> X;
8  };
9
10 //Walker class
11 template<typename T, unsigned D>
12 class Walker{
13   Vector<TinyVector<T,D>> R;//positions (AoS)
14   Buffer<T> Any; //anonymous buffer
15 };
16
17 template<typename T, unsigned D>
18 class ParticleSet{
19   using Walker_t=Walker<T,D>;
20   //Arrays of particle attributes
21   Vector<TinyVector<T,D>> R;//positions (AoS)
22   Vector<TinyVector<T,D>> G;//gradients (AoS)
23   Vector<T>               L;//laplacians
24
25   //containers of Walkers
26   Vector<Walker_t*>  Walkers;
27
28   //copy a Walker to perform a MC step
29   void loadWalker(const Walker_t& awalker) {
30     R=awalker.R;
31   }
32 };
33
34 void pseudo_qmc() {
35    using Particles=ParticleSet<double,3>;
36    Particles E;
37    Particles Ions; //shared among threads
38    TrialWaveFunction Psi(E,Ions);
39    #pragma omp parallel
40    {
41      Particles E_th(E);
42      TrialWaveFunction Psi_th(Psi)
43      #pragma omp for nowait
44      for(size_t iw=0; iw<nw; ++iw) {
45        E_th.loadWalker(*(E.Walkers[iw]));
46        for(size_t k=0; k<N; ++k) {
47          //PbyP update with DMC Algo.1
48        }
49        E_th.storeWalker(*(E.Walkers[iw]));
50      }
51    }
52 }
```

Figure 4: A simplified QMC code using OpenMP, showing a driver method `psuedo_qmc` and core abstractions for D-dimensional particle simulations. Operators and other utility methods are not shown.

known at run time and each orbital component can have any number of scalars to compute the differences before and after a move. `loadWalker/storeWalker` methods copy a `Walker` data to the compute objects for independent updates on a block of `Walker`s. High-level physics is expressed using only `ParticleSet` and `TrialWaveFunction`.

The reference implementation pre-computes and stores all the elements needed by `TrialWaveFunction` for the PbyP updates beforehand and then retrieves and modifies them during the updates. The anonymous `Buffer` holds any number of scalars to reconstruct the complete state of a `Walker` without recomputing. The memory-demanding J2 (eq. 5) keeps full N-by-N matrices for $U_2(i,j), \nabla U_2(i,j)$ (3D vector) and $\nabla^2 U_2(i,j)$ and uses minimum $5N^2$`sizeof(T)` per `Walker`. This store over compute policy was adopted when the FLOPS (sqrt, inverse, sincos) were expensive compared to reading/writing to a memory region and the number of cores per node was small (16 on BG/Q).

6.2 Performance analysis of baseline

A DMC run performs many steps $M \sim 10^6$. Either the total execution time $T_{\rm CPU}$ or the throughput which is equal to the number of MC samples generated per second, can be used as the figure of merit. For these benchmarks, we use 100-1000 steps to make runs manageable and compute the throughput as $P = M \langle N_w \rangle / T_{\rm CPU}$. Here, $\langle N_w \rangle$ denotes the average N_w. This throughput is representative of the production runs of the same target population and is directly correlated to the DMC efficiency κ. Ratios of throughputs are used to show the relative performance of the different runs on a given system and the runs on multiple systems.

For the baseline (Ref), all the quantities are in double precision by compiling QMCPACK with `QMC_MIXED_PRECISION=0`, except for the Bspline-SPO (Bspline-v and Bspline-vgh) in Fig. 2. This was the standard for production calculations prior to the version 3 release. We show performance improvements in two steps: (Ref+MP) uses mixed-precision with the reference code and (Current), mixed-precision with the final optimized code which includes all the techniques described in Sec. 7. Other intermediate steps are not presented but can be measured using different build options and miniapps.

The Ref profiles for the NiO benchmarks on KNL in Fig. 2 reveal that the computations of the distance relations among electrons (AA type) and between electrons and ions (AB type) and J2 make up close to 50% of a run. This is in contrast to the earlier profile of a smaller problem on older Harpertown quad-core processor that shows close to 50% is spent on Bspline-SPO routines [11]. We attribute these changes in the profiles to i) an increasing penalty of scalar operations using AoS data types on wide SIMD processors, ii) the high pressure on memory subsystems with more electrons in our larger benchmarks and iii) optimization of Bspline-SPO evaluations by converting critical calculations to single precision. Future problems are more demanding and the current implementation does not provide sufficient performance for practical QMC simulations of large-scale problems we would like to tackle in the future such as a disordered 1024 atom supercell of NiO.

7 TRANSFORMING QMCPACK

In order to address the performance bottlenecks identified previously we take a multi-step approach. First, we create miniapps upon which to test our algorithmic improvements. Next we change the data layout in many sections of the code, increase the use of single precision computations, and finally overhaul many distance table based algorithms involved in various parts of the code. In this section we describe these optimizations, focusing on both the methodology as well as the reasons behind the chosen algorithms.

7.1 Miniapps

We created a set of miniapps to explore solutions for the three main classes responsible for the hot-spots separately: DistTable, Jastrow (J1 and J2) and Bspline-SPO. Finally, `miniQMC` tests all the three main components. Each miniapp mimics a QMC calculation using PbyP update and non-local pseudopotentials as shown in Alg. 1 and Fig. 4. They reproduce the computational patterns, memory use, data access and thread-level parallelism of the production QMC code as realistically as possible. Command-line options are used to change the problems (N, the cutoff radius and etc) for fast prototyping, debugging and analysis.

These miniapps allow us to explore a large design space without global code modifications and to quantify any impact of the new implementations before complete integration. We use the performance model based on the theoretical analysis of QMC algorithms and empirical data on multiple platforms to project the productivity gains in real QMC simulation environments. Once we narrow down the solution space in miniapps, we implement the new data-types and methods in QMCPACK to maximize the reuse of the existing framework and to continue supporting the high-level physics abstractions that are essential for QMC method development.

7.2 Mixed precision

For the first optimization of the code, our work expands the use of single precision in the most performance critical kernels of QMCPACK including DistTable and Jastrow. We convert the key data structures and calculations to single precision, while keeping the precision-critical computation in double precision. These improvements are already available in 3.0.0 version and are enabled with `QMC_MIXED_PRECISION=1` flag. Prior to v3.0.0, only Bspline-SPO data and evaluations use single precision. This new feature significantly speeds up computations and reduces the memory usage associated with walkers and threads by half.

7.3 SoA data layout update

The object-oriented (OO) and generic programming paradigm is widely adopted for large-scale, complex HPC applications such as QMCPACK. The AoS datatypes (C++ objects) are natural choices to express mathematical concepts and logics of physics simulations. Expression templates allow optimization of complex algorithms at the compiler time. However, the abstraction penalty can be high and outweighs the benefit of using OO and generic approaches.

Production applications, such as QMCPACK, are complex often comprising millions of lines of code and support various needs of the developers and users. The SIMD-friendly solutions to accelerate the current hot-spots may harm the overall performance and limit how the high-level physics is expressed. It is essential to consider the algorithms and the balance of computations and memory access of the entire application. Any changes must be portable and extensible by the developers. This work adheres to C++11 and OpenMP 4

```
1  //Generic SoA container and key operators
2  template<typename T, unsigned D>
3  class VectorSoaContainer{
4    aligned_vector<T> X;
5    TinyVector<T,D> operator[](size_t i) const;
6    template<typename VA>
7    VectorSoaContainer<T,D>& operator=(const VA& rhs);
8  };
9
10 //Add a new data member Rsoa in SoA
11 template<typename T, unsigned D>
12 class ParticleSet{
13
14   VectorSoaContainer<T,D> Rsoa;
15
16   void loadWalker(const Walker_t& awalker) {
17     R=awalker.R;
18     Rsoa=awalker.R; //AoS-to-SoA assignment
19   }
20 };
```

Figure 5: Update to simplified QMC code in Fig. 4 with `VectorSoaContainer` and modified `ParticleSet` class. Utility functions are not shown.

SIMD standards to facilitate auto vectorization and optimizations by compilers and does not use any platform-specific optimizations, although they are not excluded for the future development.

In order to enhance SIMD efficiency, we adopt the concepts and techniques of SDLT [9, 10] and implement a generic container, `VectorSoaContainer<T,D>` (VSC), to encapsulate a vector of non-scalar types. A VSC object is a transposed form of the corresponding AoS object in SoA format and provides access and utility methods to interact with the AoS counterparts in place. The SoA containers use cache-aligned allocators chosen at the compiler time. On Intel platforms, we use the TBB cache-aligned allocator as their default allocator. Figure 5 presents important details of `VectorSoAContainer` class and its typical use in QMCPACK.

We introduce a new data member `Rsoa` in the `ParticleSet` class and implement SIMD-friendly methods in DistTable and Jastrow classes using the new SoA objects. The overhead of the duplicate containers to hold electron positions in `R[N][3]` (AoS) and `Rsoa[3][Np]` (SoA) is negligible in terms of computation and storage. Here, `Np` includes the padding for alignment. The only extra operations are the additional assignment in `loadWalker` and the update when a move is accepted.

For the k-th electron move during the PbyP update (L47 Fig. 4), all the routines are functions of the position `R[k]` of the active electron k and `Rsoa` of the electrons and ions. The computational kernels are expressed as 1-by-N and 1-by-N_{ion} relations, e. g., the distances $d(k,i) = |\mathbf{r}_i - \mathbf{r}_k|$, displacement vectors $d\mathbf{r}(k,i) = \mathbf{r}_i - \mathbf{r}_k$ and $\nabla U_2(k,i)$ for $i \neq k$. They are now implemented using the loops over N or N_{ion} that can be easily vectorized by the compilers. When a move is accepted, both `R` and `Rsoa` (6 floats), are updated with the new position for the active electron. The positions of the ions (a `ParticleSet` object) are fixed during a QMC calculation and the ions' `Rsoa` is reused throughout the calculation.

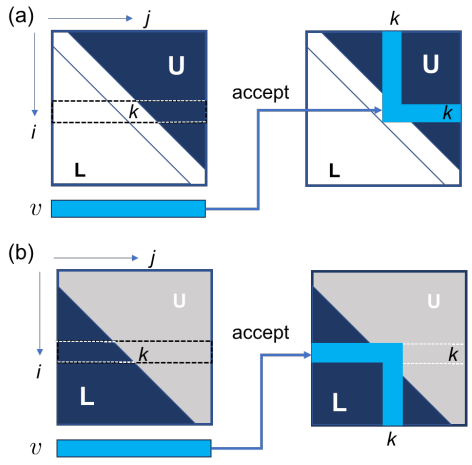

Figure 6: Schematics for AA (symmetric) distance table management (a) before and (b) after the optimizations. v is a separate array to hold the temporary data for the k-th electron move. The column update in (b) is later removed with the compute on the fly optimization.

7.4 Forward update method

Use of miniapps for the development has additional advantages over tackling the full code transformation from either the top or bottom. They expose the performance bottlenecks and inefficiencies of the current implementation that were not obvious or hidden by the primary hot-spots analysis. As we improve the SIMD efficiency with SoA containers, the cost of memory movement and the pre-compute-and-store policies turned out to be too high and diminish the benefit of the AoS-to-SoA layout transformations in the main computations.

As a particle-based method, managing the distance tables, equivalent to the nearest-neighbor lists in classical molecular dynamics codes, is critical for efficiency. The distance-table objects can be reused any number of times depending upon how many Jastrow orbitals constitute the trial wavefunction Ψ_T and what basis set is used for SPOs. They are also used by Hamiltonian objects when the measurements are made.

The top panel of Fig. 6 illustrates how the reference QMCPACK handles the electron-electron (AA) distance table. It stores the upper triangle in a packed storage as shown in dark blue and labeled as U. When the k-move is accepted, the temporary container v is copied to update U. The packed storage needs $N(N-1)/2$ scalars and requires only N copies for the update. However, the access patterns on SIMD processors are not favorable for compiler auto-vectorization due to unaligned accesses. Such inefficient data-access patterns are repeated in other routines.

We develop new algorithms to reduce memory operations, exploiting the sequential nature of the PbyP update: it performs ordered moves of N electrons and most of the properties associated with $k' < k$ electrons are not needed during the drift-and-diffusion stage. Instead of updating the full row or column of the active electron k, we update only the data necessary for the future moves and delay other computations and retain minimum quantities in memory, until the measurement is made.

The bottom of Fig. 6 presents the new method for the AA distance table. We use the full $N \times N^p$ storage (including padding) even for the AA types, increasing the memory use roughly by two. This compromise is justified as we can achieve close to the ideal speedup of vectorization from double scalars to packed floats, with cache-aligned access for each row of N^p and v. We develop a forward-update method, leaving U untouched or partially updated as the Upper triangle is not used by other methods. The k-th column update is strided by N^p but only $k' > k$ are updated upon acceptance, leaving the number of copy operations unchanged.

Similar approaches are taken in electron-ion (AB) distance table and Jastrow orbitals to eliminate unnecessary memory movements. The bulk of improvements in the Jastrow routines comes automatically with the changes in the distance tables and `ParticleSet`. Jastrow orbitals are the consumers of AA or AB distance tables and the cache-aligned, SIMD-friendly data-types allow straightforward code modifications and facilitate compilers' autovectorization.

7.5 Compute on the fly and memory savings

In addition, new algorithms [26] are developed to reduce memory used by Jastrow objects. The factorized form of Ψ_T makes the ratio computations a product of each component. The J1 contribution to the ratio Eq. (5) depends on the difference of U_k^1 for the k-th electron before and after the move, as

$$\Delta J_1 = U_k^1(\mathbf{R}') - U_k^1(\mathbf{R}),\ U_k^1 = \sum_I^{N_{\rm ion}} U_I(|\mathbf{r}_I - \mathbf{r}_k|). \quad (8)$$

Two-body Jastrow takes the similar form as $\Delta J_2 = U_k^2(\mathbf{R}') - U_k^2(\mathbf{R})$. To reuse the computed values, the Ref implementation uses three $N \times N$ matrices for the values, gradients (D=3) and Laplacians, total of $5N^2$ scalars per walker. The gradients are stored in an AoS container and both the column and row are updated for each accepted move. The SoA transformation in DistTable objects makes the Jastrow loops highly vectorizable by the compilers. With highly sped up computations, due to the single-precision use and SoA transformations, we can afford to eliminate the intermediate data all together and keep the memory use of J2 at $5N$`sizeof(T)`. We apply such compute-on-the-fly approaches whenever profitable.

Finally, we redesign `ParticleSet` and `TrialWaveFunction` member functions to clearly define the roles and requirements of the virtual functions for move, accept/reject and measurement. These changes make it possible to expand compute-on-the-fly methods to DistTable. Instead of precomputing the full table and updating the column and row as depicted in Fig. 6, we now compute the row k with the current position $\mathbf{r}_k$ before making the move. This eliminates the strided copy for the column updates. We retain $O(N^2)$ storage in DistTable, since they are used multiple times by `Hamiltonian`

objects. The results denoted as "Current" are obtained using the implementation that includes all the optimization steps discussed in this section.

8 RESULTS AND DISCUSSIONS

Figure 1 summarizes the outcome of the transformative changes of this work[2]. We present the relative performance of four sets of strong-scaling runs of a 64-atom supercell of NiO on Trinity (KNL) at LANL and Serrano (BDW) systems at SNL. The throughputs are normalized by that of the Ref code on BDW using 64 sockets (32 nodes and 1152 cores). We use 1 MPI task per KNL node (BDW socket) and two threads per core. The target DMC population is set at 131072. This corresponds to one walker per thread, on average, for the 1024-node runs on KNL. In all cases, the parallel efficiency is high, 90% (KNL) and 98% (BDW), and 2-4.5x speedup is obtained through the optimizations.

All the performance improvements in Fig. 1 are attributed to the data-layout transformations, reduced memory operations and the expanded use of single precision. The MPI communications are the same for both Ref and Current code: allreduce to compute running averages for E_L and other global properties and send/recv of serialized `Walker` objects during the load-balancing steps. The memory-reduction algorithms in Jastrow reduce the `Walker` message size by 22.5 MB for the NiO-64 problem. There is, however, no fundamental change in communications that still have low overhead. Therefore, we focus on the single-node benchmarks and provide comprehensive analysis of the on-node performance to show the impact of our work on performance, memory footprint and energy consumption for the rest of the section.

8.1 Roofline and hot-spot analysis

The hot-spot profile and roofline [27, 28] analysis of the 32-atom NiO supercell in Fig. 7 shows the evolution of the four major kernels before and after the optimizations on BDW. Similarly, Fig. 2 shows hot-spot profiles of NiO benchmarks on KNL. The transformations significantly decrease the time spent in DistTable, J2 and Bspline-vgh. Other determinant-related computations, SPO-vgl and DetUpdate, are sped up by more than two with the double-to-single transition in A^{-1}. The roofline performance model on BDW shows large jump in both AI and FLOPS with the Current code. Efficient vectorization is enabled in DistTable, Jastrow, Bspline-vgh and SPO-vgl with the SoA data-types. The greater increase in AI and FLOPS of DistTable and J2 is the combined effect of the expanded use of single precision and the improved data

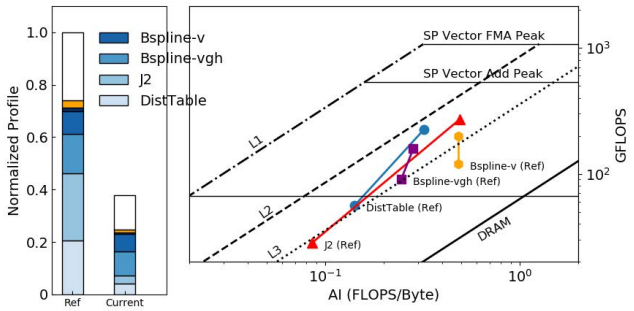

Figure 7: Normalized hot-spot profile and roofline analysis of NiO-32 on BDW. Current version hot-spot profile accomodates the speedup wrt. Ref version.

structures and algorithms. Bspline-v kernel is unchanged but its efficiency increases with the memory optimizations on BDW.

Our optimizations have quantitatively different effects on BDW and KNL processors due to their architectural differences. KNL has twice the single precision SIMD width of BDW's, making the theoretical vectorization speedup twice as large. The bandwidth of 16 GB MCDRAM in flat mode is about 8 times higher than that of one-socket BDW. The cache subsystems, their sizes and associativities, are also different. The shared L3 cache on BDW can make up for the low DDR bandwidth: Fig. 7 shows that all four kernels lie above the L3 roofline after the optimizations. Despite these architectural differences, qualitative impacts of the optimizations are the same for both processors.

The data-layout transformation enables close to the ideal speedup in DistTable computations, due to its contiguous stream of data access. For Jastrow routines, the vectorization efficiency is slightly lower due to the branch conditions originated from the finite cutoff of the Jastrow functors in Fig. 3. Compute-on-the-fly policy in Jastrow routines are critical as we eliminate all $\mathcal{O}(N^2)$ memory storage. The only remaining $\mathcal{O}(N^2)$ storage per walker comes from the determinant objects in storing A^{-1}. All optimizations of Current work result in 5x (DistTable), 8x (Jastrow), 1.7x (Bspline-vgh) and 1.3x (Bspline-v) speedups for the NiO-32 benchmark on BDW. Figure 2 shows the normalized profiles of both NiO benchmarks on KNL, showing similar speedups for each routine.

8.2 Benchmark results and discussion

Having established the efficiency and scalability of QMCPACK with our current methods, we turn to the detailed performance analysis of NiO benchmarks on single KNL and BDW processors. The system details are provided in Sec. 5. To keep the the amount of work similar, we use the target population of 1024 (KNL) and 1040 (BDW), equivalent to 8 and 24 *average* walkers per thread on KNL and BDW, respectively.

[2] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.
Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

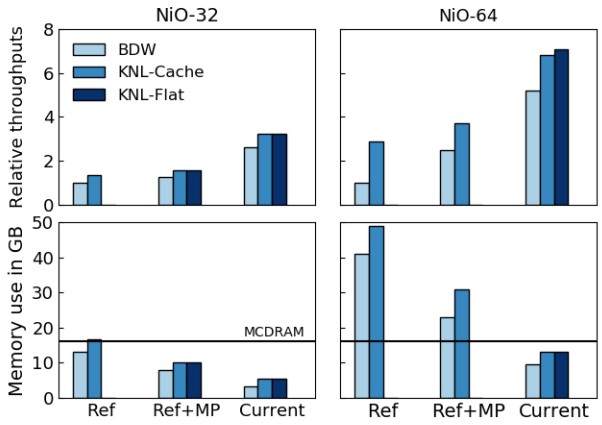

Figure 8: Speedup and memory-usage reduction of NiO benchmarks. The throughputs are normalized by those of Ref on BDW.

Our hyperthreading study of the 32-atom NiO supercell benchmark with the optimized Current version shows its positive impact on both BDW and KNL processors. This is expected because hyperthreading can hide latency in memory-intensive operations such as Bspline-SPO routines. They are memory-latency sensitive due to random accesses of the 4-dimensional read-only table and are also memory-bandwidth limited. Using 2 threads per core provides 10% and 8.5% throughput improvements for BDW and KNL respectively. On KNL, using 2 threads per core is optimal for this system and using 3 or 4 threads per core does not improve the throughput. This is generally true for other problems we have investigated.

Figure 8 shows performance and memory usage of NiO on BDW and KNL in cache and flat memory modes for the Ref, Ref+MP and Current. The missing data of KNL-flat is due to the memory footprint being more than 16 GB, the capacity of the MCDRAM. The throughputs are normalized by the Ref on BDW and higher throughput means higher performance.

Mixed-precision implementation (Ref+MP) reduces memory bandwidth usage by storing the key datasets in single precision and accelerates bandwidth-bound routines. However, it does not benefit functions with low SIMD efficiency, limiting its impact on KNL. The 64-atom supercell of NiO doubles the problem in size and therefore, its computational cost and memory use are accordingly higher. It is expected to be bandwidth bound and gains more by MP than smaller problems. The speedups on KNL, 1.3x of NiO-64 compared 1.16x of NiO-32 support this performance projection. Expanded use of single precision, together with the help of shared L3 on BDW, lowers the bandwidth demands for both the benchmarks and leads to higher speedups, 2.5x (NiO-64) and 1.3x (NiO-32).

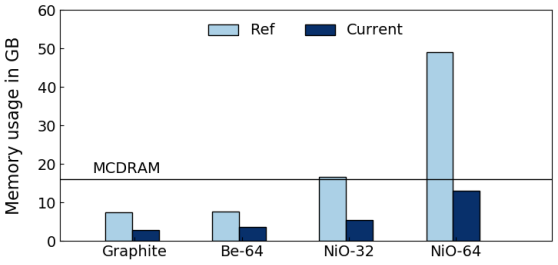

Figure 9: Memory usage on KNL processor.

The performance of Current runs are more than doubled on both BDW and KNL compared to Ref+MP. More importantly, the memory usage has gone down dramatically as much as 36 GB from Ref for the NiO-64 benchmark, allowing all the benchmarks to run on KNL in flat mode. The performance gains from cache to flat mode are modest, around 3% for NiO-64. The importance of high BW is evident for the NiO benchmarks. Exclusive use of DDR (`numactl -m 0`) slows down the Current by 5.4x for NiO-64, which is commensurate with the stream bandwidth difference of MCDRAM and DDR on KNL. The low BW of DDR affects the 32-atom supercell of NiO less, slowing it down only by 2.3x, as the compute-bound routines play greater roles for the smaller problems.

The bottom of Fig. 8 shows the measured memory usage in GB on BDW and KNL for NiO problems. The memory footprint of Ref QMCPACK grows as $\gamma(N_{th} + N_w)N^2$ excluding the read-only 3D B-spline table which is shared by all the threads. Here N_{th} and N_w represent the number of threads and walkers respectively. The pre-factor γ depends on the details of Ψ_T and the minimum is 60 bytes to store J2 and determinant objects in double precision. This allocation policy is the design choice to make thread-level parallelization efficient by maximizing the data locality and removing data racing conditions. Separating `Walkers` from the compute engines, `ParticleSet` and `TrialWaveFunction`, makes it possible to use an arbitrary number of `Walkers` per node and any number of nodes under the memory constraints.

The increased thread-level parallelisms on newer processors, such as KNL, limit the problems we can solve using the Ref implementation. We expect that the simulations of 1000s of electrons will become the norm, rather than an unusual scenario in the near future. Reducing memory footprint is critical, while utilizing all the resources available on a node for the productivity. Figure 9 shows $\mathcal{O}(N^2)$ memory savings on the four benchmarks in Current through the use of new algorithms and expanded use of single precision. For instance, 36 GB reduction in memory is achieved for NiO-64 and the total memory footprint is less than 16 GB, the memory capacity of a BG/Q node. Such savings open up new opportunities for the scientists and allow them to study the problems they cannot solve with Ref QMCPACK today.

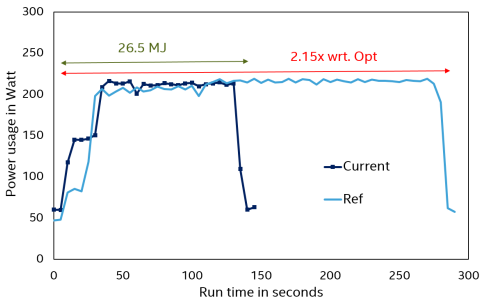

Figure 10: Energy usage of NiO-32 benchmark on KNL.

Figure 10 shows energy reduction after the optimizations (Current) compared to Ref for NiO-32. Power usage is plotted against the time of execution. The Ref version required to use MCDRAM in cache mode as mentioned earlier. Power is measured with the turbostat Linux utility with a 5 second interval. We add PkgWatt and RAMWatt values, which represent the total power used by the CPU+MCDRAM and DDR. Power usage fluctuate within the range of 210-215 watt during the DMC phase for both Ref and Current. A similar power profile was obtained for the larger NiO-64 problem on KNL. Excluding the initialization and warmup time, the energy reduction is roughly equal to the speedup obtained with the optimizations. This means huge energy savings for the production simulations running on 1000's of nodes for hours and days as well as huge productivity gains in science.

8.3 Performance summary and portability

The improved performance from our work is not just limited to BDW and KNL. As pointed out, no platform-specific optimizations or intrinsics are used in Current and we employ the standard features modern C++ compilers support. Table 2 gives the final speedups of the four benchmarks on BG/Q, BDW, and KNL processors. The speedup compares the performance of the Current implementation over the Ref code on each system and does not reflect the absolute performance of different processors. These benchmarks are distinct in their sizes and computational characteristics due to the different constituent ions and the cell shapes. They exercise different code paths determined at run time for each benchmark. The compiler's support for C++11, OpenMP SIMD and their abilities to produce optimized binaries vary. Nevertheless, we are able to accelerate the entire QMC simulations across the platforms and the problems.

Table 2: Speedup of Current over Ref on BG/Q, BDW and KNL, respectively.

	Graphite	Be-64	NiO-32	NiO-64
BG/Q	1.6	1.3	1.3	2.4
BDW	2.9	3.4	2.6	5.2
KNL	2.2	2.9	2.4	2.4

8.4 Outlook and future work

The much improved efficiencies of the top hot-spots increase the importance of the other kernels that have not been addressed so far, including DetUpdate for A^{-1} update. Figure 2 shows DetUpdate is 10 % for NiO-64 using Current, as opposed to 7 % with Ref. The asymptotic $\mathcal{O}(N^3)$-scaling of QMC methods arises from DetUpdate based on Sherman-Morrison formula. For the current problems on CPUs with multiple cache levels and ample capacity, the computations are dominated by DistTable, Jastrow, and SPO evaluations and grow as $\mathcal{O}(N^2)$. However, as the system size grows, DetUpdate using BLAS2 becomes increasingly important and becomes the bottleneck of QMC calculations.

Several alternatives based on Woodbury matrix identity [29], the generalization of Sherman-Morrison formula, can be applied to DetUpdate. One promising solution is a *delayed-update* scheme designed to evaluate multiple accepted moves before any updates are made to A^{-1} [30]. The delay factor can be adjusted to optimize the performance of higher BLAS functions for the update and the ratio computations for any size N. No structural changes are required to implement these new DetUpdate methods.

The efficient vectorization and reduction in memory footprint, e. g., to run NiO-64 using 128 threads entirely on 16 GB MCDRAM in flat KNL memory mode, is critical to solve today's problems faster and to enable simulations of much bigger and demanding future problems. The transformations presented in this work increase the science productivity and resource utilization of the systems we have and are the critical step to future-proof QMCPACK for the systems we will have.

Let's consider a 512-atom supercell of NiO (6144 electrons). It is 8 times bigger than the current NiO-64 and would take 512 times longer per step and require 64 times more memory with the Current, even with all the optimizations. Exposing extra parallelisms is the only path to make QMC practical to study such problems. BLAS3 routines are highly optimized and parallelized on any platform. The "fat" loops over the electrons and ions are ideally suited to parallelize the computations for each walker.

Our previous work [8] demonstrated that tiling of the big B-spline table and parallel execution over the array-of-SoA (AoSoA) objects can reduce the time to complete a QMC step. We propose to extend those ideas to full QMCPACK. Its object-oriented designs are amenable for either nested loop parallelization or any task-based parallelism. OpenMP standards support various ways to implement the parallel executions. Which solution will provide the most productive path for the science is unknown. We expect our approaches based on miniapps and iterative transformation processes to facilitate future developments as well.

9 CONCLUSIONS

We presented single-node optimizations for QMCPACK, a leading US-DOE quantum Monte Carlo application, and demonstrated the transferability of these optimizations to

highly parallel runs on multiple platforms. A set of miniapps representing the computational and data access patterns of QMC were developed for fast exploration, debugging and evaluations. We applied the structural changes in QMCPACK by introducing new abstractions in the SoA format and implementing the methods that can be optimized by modern C++ compilers. Our work systematically expanded the use of single precision to reduce memory bandwidth demands and footprint, while preserving the fidelity of double-precision calculations. Taking advantage of the increased SIMD efficiency of the new kernels, we developed and implemented new algorithms to further improve the performance and to reduce the memory footprint. All these are seamlessly incorporated in the production QMCPACK in portable and maintainable ways to increase the productivity of the developers and users.

ACKNOWLEDGMENTS

We thank Jason Sewall, Roland Schulz, Victor Lee, John Pennycook, and Dayle Smith for their helpful discussions and reviewing this manuscript. We also thank Intel® Advisor team for providing timely engineering builds and quick responses. This work is supported by Intel Corporation to establish the Intel Parallel Computing Center at Argonne National Laboratory. LS was supported by the Advanced Simulation and Computing - Physics and Engineering models program at Sandia National Laboratories. AB was supported through the Predictive Theory and Modeling for Materials and Chemical Science program by the Office of Basic Energy Science (BES), Department of Energy (DOE). Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under Contract No. DE-AC04-94AL85000. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract No. DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] 4 applications sustain 1 petaflop on Blue Waters. (2013). http://www.ncsa.illinois.edu/News/Stories/PFapps/ http://www.ncsa.illinois.edu/News/Stories/PFapps/.

[2] QMCPACK. http://www.qmcpack.org

[3] Jeongnim Kim, Kenneth P Esler, Jeremy McMinis, Miguel A Morales, Bryan K Clark, Luke Shulenburger, and David M Ceperley. 2012. Hybrid algorithms in quantum Monte Carlo. *Journal of Physics: Conference Series* 402, 1 (2012), 012008. http://stacks.iop.org/1742-6596/402/i=1/a=012008

[4] Ye Luo, Anouar Benali, and Vitali Morozov. 2015. Accelerating the B-Spline Evaluation in Quantum Monte Carlo. (2015). http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post337.html

[5] Todd L. Veldhuizen. 2004. *Active Libraries and Universal Languages*. Ph.D. Dissertation. Indianapolis, IN, USA. AAI3134053.

[6] Matthias Müller. 2000. Abstraction benchmarks and performance of C++ applications. *Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications* (2000).

[7] CORAL collaboration, Benchmark Codes. http://https://asc.llnl.gov/CORAL-benchmarks/

[8] Amrita Mathuriya, Ye Luo, Anouar Benali, Luke Shulenburger, and Jeongnim Kim. 2016. Optimization and parallelization of B-spline based orbital evaluations in QMC on multi/many-core shared memory processors. *IPDPS 2017 proceedings* abs/1611.02665 (2016). http://arxiv.org/abs/1611.02665

[9] Introduction to the SIMD Data Layout Templates. https://software.intel.com/en-us/node/684050

[10] Nimisha Raut, Alex Wells, and George Raskulinec. 2016. Data Layout Optimization Using SIMD Data Layout Templates. (2016). https://software.intel.com/en-us/articles/data-layout-optimization-using-simd-data-layout-templates

[11] Kenneth P Esler, Jeongnim Kim, Luke Shulenburger, and David M Ceperley. 2012. Fully accelerating quantum Monte Carlo simulations of real materials on GPU clusters. *Computing in Science and Engineering* 14 (2012), 40. http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.122

[12] Jeongnim Kim and QMCPACK developers. QMCPACK manual. https://github.com/QMCPACK/qmcpack/raw/develop/manual/qmcpack_manual.pdf

[13] Ye Luo et al. 2017. Computing with reduced precision in QMC. *In preparation* (2017).

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[15] ISO/IEC 14882:2011, Informataion technology – Programming langauges C++. https://www.iso.org/standard/50372.html

[16] W. M. C. Foulkes, L. Mitas, R. J. Needs, and G. Rajagopal. 2001. Quantum Monte Carlo simulations of solids. *Rev. Mod. Phys.* 73 (2001), 33. Issue 1. https://doi.org/10.1103/RevModPhys.73.33

[17] Gary D. Knott. 2000. *Interpolating cubic splines*. Springer Science & Business Media.

[18] George E P Box and Jenkins Gwilym M. 1976. *Time Series Analysis : Forecasting and Control*. Holden-Day.

[19] S. Fahy, X. W. Wang, and Steven G. Louie. 1990. Variational quantum Monte Carlo nonlocal pseudopotential approach to solids: Formulation and application to diamond, graphite, and silicon. *Physical Review B* 42, 6 (1990), 3503. https://doi.org/10.1103/PhysRevB.42.3503

[20] Bryan K Clark, Miguel A Morales, Jeremy McMinis, Jeongnim Kim, and Gustavo E Scuseria. 2012. Computing the energy of a water molecule using multideterminants: A simple, efficient algorithm. *The Journal of Chemical Physics* 135, 24 (2012), 244105. https://doi.org/doi:10.1063/1.3665391

[21] Intel® Parallel Studio XE 2017. https://software.intel.com/en-us/intel-parallel-studio-xe

[22] Intel compiler options: -O3 -ip -restrict -unroll -g -debug inline-debug-info -openmp -std=c++11", with -axCORE-AVX2,AVX (BDW) and -xMIC-AVX512 (KNL).

[23] Intel® VTune™ Amplifier 2017. https://software.intel.com/en-us/intel-vtune-amplifier-xe

[24] Intel® advisor 2017. https://software.intel.com/en-us/intel-advisor-xe

[25] Hal Finkel. 2014. bgclang: Creating an Alternative, Customizable, Toolchain for the Blue Gene/Q. (November 16–21 2014).

[26] Ye Luo et al. 2017. Fast evaluation of Jastrow factors in QMC . *In preparation* (2017).

[27] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Commun. ACM* (2009). https://doi.org/10.1145/1498765.1498785

[28] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2014. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters* 13, 1 (2014), 21–24.

[29] Max A Woodbury. 1950. *Inverting modified matrices. Memorandum Report 42, Statistical Research Group*. Princeton, New Jersey, USA.

[30] Tyler McDaniel, Ed D'Azevedo, Ying Wai Li, Paul Kent, Ming Wong, and Kwai Wong. 2016. Delayed Update Algorithms for Quantum Monte Carlo Simulation on GPU: Extended Abstract. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale (XSEDE16)*. ACM, New York, NY, USA, Article 13, 4 pages. https://doi.org/10.1145/2949550.2949579