Field-Programmable Gate Arrays and Quantum Monte Carlo: Power Efficient Co-processing for Scalable High-Performance Computing

Salvatore Cardamone,[1, *] Jonathan R. R. Kimmitt,[1, 2] Hugh G. A. Burton,[1]
Timothy J. Todman,[3] Shurui Li,[3] Wayne Luk,[3] and Alex J. W. Thom[1]

[1] *University Chemical Laboratory, Lensfield Road, Cambridge CB2 1EW, United Kingdom*
[2] *Department of Computer Science and Technology, William Gates Building,*
J. J. Thomson Ave, Cambridge CB3 0FD, United Kingdom
[3] *Department of Computing, Imperial College London,*
180 Queen's Gate, London SW7 2RH, United Kingdom
(Dated: October 21, 2018)

Massively parallel architectures offer the potential to significantly accelerate an application relative to their serial counterparts. However, not all applications exhibit an adequate level of data and/or task parallelism to exploit such platforms. Furthermore, the power consumption associated with these forms of computation renders "scaling out" for exascale levels of performance incompatible with modern sustainable energy policies. In this work, we investigate the potential for field-programmable gate arrays (FPGAs) to feature in future exascale platforms, and their capacity to improve performance per unit power measurements for the purposes of scientific computing. We have focussed our efforts on Variational Monte Carlo, and report on the benefits of co-processing with a FPGA relative to a purely multicore system.

I. INTRODUCTION

Quantum Monte Carlo (QMC) encompasses a class of techniques for approximating expectation values to quantum mechanical observables for many-electron systems[1]. By casting the time-independent Schrödinger equation into integral form, the high-dimensional integration — intractable through quadrature techniques — can be realised through a stochastic sampling of the many-electron wavefunction.

Relative to deterministic counterparts, QMC offers the potential to combine the favourable scaling of mean-field techniques and the accuracy of post-Hartree Fock methods[2]. Two forms of QMC in particular have gained traction as popular *ab initio* methodologies[3]: Variational Monte Carlo (VMC) and Diffusion Monte Carlo (DMC). Commonly, VMC is used to variationally optimise free parameters in the wavefunction, with DMC being subsequently applied to compute production level observables for the molecular system in question.

Prior to the failure of Dennard scaling[4], application developers were able to indulge in a "free lunch", whereby they could exploit the continual increase in processor clock frequencies with new generations of hardware to directly accelerate an application. Since on-chip power densities no longer scale down with transistor density, manufacturers instead look towards heat dissipation technologies in an effort to accommodate Moore's law[5]. However, such techniques have limitations and eventually require manufacturers to reduce on-chip voltages, forecasted to eventually culminate in an era of "dark silicon"[6].

To address the unsustainability of frequency scaling, the tendency has been for new generations of hardware to increase the number of parallel compute units on a single chip. As a result, the onus falls onto the software developer to exploit either operations or tasks that can be undertaken in parallel within an application. Furthermore, with the advent of distributed memory systems and graphical processing units (GPUs) at contemporary high-performance computing (HPC) facilities, the software developer has an enormous arsenal of hardware solutions for exploiting parallelism within an application.

For QMC methodologies, the associated embarrassing parallelism lends itself well to modern parallel architectures. There is a significant body of work demonstrating the near-linear scaling of performance with respect to the available hardware concurrency[7, 8]. More recently, GPUs have been utilised as co-processors to deliver impressive performance gains[9]. However, "scaling out" is not a sustainable solution to simulating increasingly complex systems. The power consumption associated with distributed memory systems and GPUs does not align with modern energy policies. Indeed, a consensus appears to have emerged[10–12] that the power consumption of HPC is one of the most prodigious barriers to attaining exascale levels of performance.

Field-Programmable Gate Arrays (FPGAs) are appealing candidates as processing units for novel exascale platforms[13, 14]. These devices combine low power consumption with the capacity to exploit both data and task parallelism, and are therefore applicable to a wider set of applications than solutions relying on data parallelism alone. To date, use of FPGAs in electronic structure theory has been somewhat limited, although several works demonstrate the power of this platform to accelerate scientific applications within this domain.[15–17]

* sc2018@cam.ac.uk

In this work, we look to port the compute-intensive portions of a VMC calculation to an FPGA and assess the performance relative to a CPU-bound application. We elaborate on the optimisations incorporated into our design to optimise the implementation for the purposes of co-processing. Finally, we demonstrate that our application benefits from the use of FPGAs in terms of both raw compute performance and power consumption.

II. VARIATIONAL MONTE CARLO

The following section is not intended to present an exhaustive exposition of techniques within QMC. Rather, the reader is directed to the excellent review of Foulkes and coworkers[1] for further details.

A. Implementation

Consider a molecular system comprising N nuclei and n electrons, where $\boldsymbol{R}$ and $\boldsymbol{r}$ denote their collective position vectors respectively. The time-independent electronic wavefunction of this system is represented by $\Psi(\boldsymbol{r}; \boldsymbol{R})$, where the parametric dependence on $\boldsymbol{R}$ arises from the application of the Born–Oppenheimer approximation. Expanding $\Psi(\boldsymbol{r}; \boldsymbol{R})$ in terms of the exact (orthonormalised) eigenfunctions of the molecular Hamiltonian, $\hat{\mathcal{H}}(\boldsymbol{r}, \boldsymbol{R})$, the energy of the system is determined by the expectation value $\langle \hat{\mathcal{H}}(\boldsymbol{r}, \boldsymbol{R}) \rangle_\Psi$. Alternative (time-independent) physical quantities can be obtained by substituting the molecular Hamiltonian for the appropriate corresponding operator.

From the variational principle it can be shown that for any approximate wavefunction, $\Psi_T(\boldsymbol{r}; \boldsymbol{R})$, referred to as the "trial wavefunction", the expectation value $\langle \hat{\mathcal{H}}(\boldsymbol{r}, \boldsymbol{R}) \rangle_{\Psi_T}$, provides an upper bound to the true energy of the system. Consequently, the variational principle provides an important metric for quantifying the quality of an approximate trial wavefunction, with lower energies implying a higher quality wavefunction. Herein, all explicit functional dependencies are omitted unless a new quantity is introduced.

Through the time-independent Schrödinger equation, the energy associated with a particular trial wavefunction can be expressed as

$$E_T = \frac{\int \mathrm{d}\boldsymbol{r}\, \Psi_T^* \hat{\mathcal{H}} \Psi_T}{\int \mathrm{d}\boldsymbol{r}\, \Psi_T^* \Psi_T} \,, \tag{1}$$

where integration is performed over the $3n$ electronic degrees of freedom. With some foresight, multiplication by the identity Ψ_T / Ψ_T casts this expression into a form amenable to solution through stochastic methods:

$$E_T = \frac{\int \mathrm{d}\boldsymbol{r}\, |\Psi_T|^2 \frac{\hat{\mathcal{H}} \Psi_T}{\Psi_T}}{\int \mathrm{d}\boldsymbol{r}\, |\Psi_T|^2} \,. \tag{2}$$

The above "importance sampled form" utilises $|\Psi_T|^2$ as a probability density function from which samples of the "local energy", $E_L(\boldsymbol{r}) = \Psi_T^{-1} \hat{\mathcal{H}} \Psi_T$, can be drawn. Through Monte Carlo, the resultant average of the local energy over N_{MC} (uncorrelated) samples is asymptotic to the variational energy of the trial wavefunction:

$$E_T \sim \lim_{N_{\mathrm{MC}} \to \infty} \frac{1}{N_{\mathrm{MC}}} \sum_{i=1}^{N_{\mathrm{MC}}} E_L(\boldsymbol{r}^{(i)}) \,, \tag{3}$$

where $r^{(i)}$ denotes a sample, and equality emerges in the limit $N_{\mathrm{MC}} \to \infty$.

Variational Monte Carlo (VMC) provides a means for implementing the above stochastic process. Considering an ensemble of "walkers", each representing a discrete sample of Ψ_T with a particular electronic configuration in position space, the simulation proceeds by stochastically propagating each walker through configuration space. By randomly displacing an electronic configuration, $\boldsymbol{r}' \leftarrow \boldsymbol{r}$, the Metropolis–Hastings algorithm can be invoked to accept the step according to the transition probability

$$P(\boldsymbol{r}' \leftarrow \boldsymbol{r}) = \min \left[1, \left| \frac{\Psi_T(\boldsymbol{r}')}{\Psi_T(\boldsymbol{r})} \right|^2 \right] \,. \tag{4}$$

A basic overview of the VMC algorithm follows in Algorithm 1.

Algorithm 1 Variational Monte Carlo

> **for** iMC $= 1, \ldots, N_{\mathrm{MC}}$ **do**
>   **for** iWalker $= 1, \ldots, N_w$ **do**
>   **for** iEl $= 1, \ldots, n$ **do**
> $\boldsymbol{r}'_{\mathrm{iEl}} \leftarrow \boldsymbol{r}_{\mathrm{iEl}} + \boldsymbol{\delta}$
>   Compute $\Psi_T(\boldsymbol{r}')$
>   **if** $|\Psi_T(\boldsymbol{r}')/\Psi_T(\boldsymbol{r})|^2 > \mathcal{U}(0, 1)$ **then**
>   Compute $\nabla^2_{\boldsymbol{r}'} \Psi_T(\boldsymbol{r}')$
>   Update $\Psi_T^{-1}(\boldsymbol{r}')$
>   Accumulate $E_L(\boldsymbol{r}')$
> $\boldsymbol{r}_{\mathrm{iEl}} \leftarrow \boldsymbol{r}'_{\mathrm{iEl}}$
>   **return** E_T

B. The Trial Wavefunction

Since the many-electron wavefunction is unknown in closed-form for all but the most trivial of systems, approximations must be invoked. Owing to the antisymmetry of the fermionic wavefunction, a determinant is a particularly appropriate mathematical form for the trial wavefunction. Subject to the spin-invariance of the operator whose associated observable is to be computed, the trial wavefunction can be written as the product of α- and β-spin components,

$$\Psi_T(\boldsymbol{r}^\alpha, \boldsymbol{r}^\beta) = \det[\mathcal{D}^\alpha(\boldsymbol{r}^\alpha)] \times \det[\mathcal{D}^\beta(\boldsymbol{r}^\beta)], \tag{5}$$

where $\mathcal{D}(\boldsymbol{r})$ is referred to as the Slater matrix, and $\boldsymbol{r}^\alpha, \boldsymbol{r}^\beta$ denote the set electron position with the appropriate

spin. The Slater matrix is composed of a set of one-particle molecular orbitals, $\psi(\boldsymbol{r})$,

$$\mathcal{D}^\omega(\boldsymbol{r}^\omega) = \begin{bmatrix} \psi_1(\boldsymbol{r}_1^\omega) & \dots & \psi_{n_\omega}(\boldsymbol{r}_1^\omega) \\ \vdots & \ddots & \vdots \\ \psi_1(\boldsymbol{r}_n^\omega) & \dots & \psi_{n_\omega}(\boldsymbol{r}_n^\omega) \end{bmatrix}, \qquad (6)$$

where ω symbolises an arbitrary spin-state. A molecular orbital is constructed as a linear combination of atomic orbitals (LCAO),

$$\psi_i(\boldsymbol{r}) = \sum_{j=1}^{N_{\mathrm{AO}}} c_{ij}\phi_j(\boldsymbol{r}), \qquad (7)$$

where $\phi(\boldsymbol{r})$ denotes an atomic orbital, N_{AO} is the number of atomic orbitals in the expansion and $\{c_{ij}\}$ are the expansion coefficients for the ith molecular orbital. The atomic orbital is formed from a linear combination of N_p primitive functions,

$$\phi_j(\boldsymbol{r}) = f(\ell_j, m_j, |\boldsymbol{r} - \boldsymbol{R}_j|) \sum_{k=1}^{N_p} d_{jk} \exp(-\zeta_k |\boldsymbol{r} - \boldsymbol{R}_j|^2)$$

$$= f(\ell_j, m_j, \boldsymbol{r}) \sum_{k=1}^{N_p} \eta_{jk}(\boldsymbol{r}). \qquad (8)$$

In the above, $\boldsymbol{R}_j$ is the position vector of the atom to which the atomic orbital is centred, ζ_k is the width of the gaussian function and d_{jk} are the expansion coefficients for the jth atomic orbital. $f(\ell_j, m_j, \boldsymbol{r})$ is a function of the azimuthal and colatitudinal quantum numbers associated with the atomic orbital, i.e. its angular momentum.

It is common practice to multiply the above determinantal form of the wavefunction with a function of inter-particle distances, referred to as the Jastrow factor, to account for the correlation effects between particles[18]. While inclusion of the Jastrow factor is one of the powerful features of QMC methods in general, we omit discussion of it in our work for the sake of simplicity. Furthermore, the sum of gaussian functions arising in the expression for the molecular orbital is often substituted for a cubic spline to ease the computational burden of complexity scaling with the number of atomic orbitals in the system[19]. We avoid use of these splines to ensure our application remains compute bound.

We wish to stress that our resultant implementation is optimised for a subset of VMC capabilities. Rather than claim our results are representative of all VMC calculations, we hope this work serves to illustrate the potential benefits of using FPGAs, and as such should be considered exploratory as opposed to all-encompassing.

III. FPGAS

A. Basics and Nomenclature

From the beginning it is worth clarifying the distinction between **latency** and **throughput**, the former being the time taken to traverse a computational workflow, and the latter being the number of outputs from the workflow per unit time; for the processing of a single data item, these are just reciprocally related. For multiple data items, pipeline parallelism offers the capacity to increase computational throughput at the cost of latency. The rationale behind leveraging latency for throughput is best illustrated by example, for which we will refer to Figure 1.

Consider a "stream" of data, $x_0, x_1, \dots$, where the subscript enumerates order. In Figure 1a, we observe an unpipelined implementation of three data-dependent modules: f, g and h, each delivering a result through composition with the preceding modules, i.e. $f(x)$, $g(x, f)$ and $h(x, f, g)$, where nested parentheses are omitted for clarity. Each module has an associated latency, ℓ_f, ℓ_g, ℓ_h. The latency of this workflow is given by the longest route through the workflow, so here is simply the sum of the individual module latencies. The maximum frequency at which a clock can drive the workflow is determined by the propagation delay associated with the longest pathway in the workflow. As the module latencies differ, allowing multiple elements of the data stream to concurrently reside within the workflow results in improper behaviour. The occupancy of data within modules of the workflow with respect to time is depicted in the right-hand side of Figure 1a.

Figure 1b depicts the same workflow as that in 1a, but with the addition of registers (red boxes and their associated latencies) capable of storing data, across the workflow. As such, each module effectively inherits its latency from the maximum module latency in the workflow, ℓ_g in this case. However, introduction of the registers allows the workflow to process multiple data concurrently. In other words, the data stream exhibits temporal, or **pipeline**, parallelism, i.e. the task can be executed as a cascade of sub-tasks[20]. The resultant pipelined implementation then increases the throughput of the application, at the cost of a delay in processing a single datum.

An FPGA grants the application developer the means to realise a computational pipeline spatially in silicon. The FPGA is a matrix of configurable logic blocks (CLBs), fundamental programmable units comprising a lookup table and flip-flop (a logic unit capable of storing a state), amongst other logical units depending upon the chip. Signals are routed through the CLBs by a series of programmable switches, theoretically permitting the transmission of a signal between any two CLBs (although in practice one would wish to co-localise the connected CLBs for an optimal configuration).
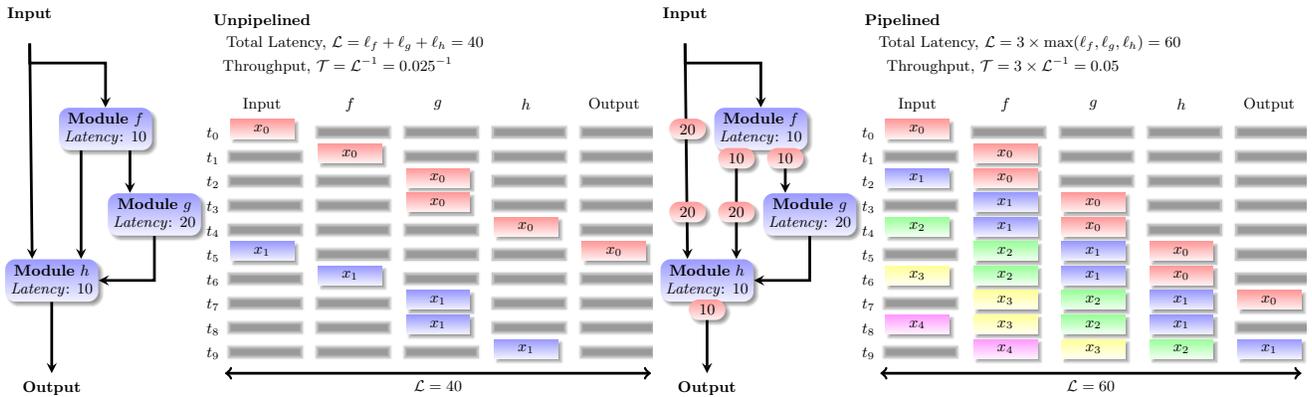
(a) Unpipelined three-module workflow, each module being dependent upon the output of the former in sequence. A single data element must traverse the entire workflow prior to the entry of the next data element. While the latency is comparatively lower than for a pipelined case, the throughput is also low.

.

(b) Pipelined three-module workflow, each module being dependent upon the output of the former in sequence. Multiple data elements are allowed to reside within the workflow simultaneously. The latency of the workflow is higher than the unpipelined case owing to the necessity that all modules have associated registers to increase the *effective* module latency to that of the highest module latency. However, throughput is twice as high as for the unpipelined case, with a new result output every ℓ_g once the pipeline is filled.

FIG. 1: An application comprising a series of data-dependent modules and its amenability to optimisation through pipeline parallelism.

Typically, a number of "hard blocks" (such as digital signal processors, dedicated floating point units, static random access memory (SRAM) blocks, etc.) are also available on the chip for specialised tasks that may be costly to implement directly from CLBs.

A configuration for the FPGA is loaded at runtime into a flash memory. The contents of this memory are used to configure the programmable switches, routing signals as per the uploaded configuration. For the types of computation considered in this work, the FPGA is able to interact with a general purpose processing unit through, for instance, a PCIe connection. Since the FPGA is configured to perform a specific set of computations, the overheads associated with general purpose computing (such as scheduling and interrupts controlled by an operating system) are eliminated. Furthermore, since the clock frequency at which an FPGA configuration can be run is dictated by the propagation delay across the chip, the implementation will be clocked at far lower frequency than a CPU or GPU. These two factors result in a dramatic decrease in the power consumption of an FPGA. As a rough indication, an FPGA consumes roughly an order of magnitude less power than a CPU or GPU. As such, FPGAs are attractive devices for the purposes of low-power computation.

Naturally there are some significant difficulties in the use of FPGAs. The conventional means for programming FPGAs requires knowledge of low-level unabstracted hardware description languages (HDLs). Use of these languages requires expertise in low-level design, and is therefore typically inaccessible to application developers from the physical sciences. Consequently, development times for FPGA-based applications are enormously lengthy.

However, there exist a number of tools available to the developer for porting a complex application to FPGAs. High Level Synthesis (HLS) tools, such as those marketed by the major FPGA vendors Xilinx and Intel/Altera, offer the capacity to annotate high-level source code (C/C++ and OpenCL predominantly) with preprocessor directives, subsequently used to construct a HDL implementation of the application. Nevertheless, high performance implementations require considerable restructuring relative to a conventional CPU-based alternative to facilitate the inference of a pipelined implementation by the HLS. While the level of expertise required by the developer is reduced, a significant understanding of the platform is imperative.

An alternative approach is to use an embedded domain specific language (EDSL), providing a library for a high-level host language to support FPGA-based constructs, such as streams of data. The implementation is analysed to construct an abstract syntax tree, which can subsequently be used to generate the HDL implementation of the application. An example of such a facility is the extension to java provided by Maxeler Technologies, of which more will be said later in this work. Similar to HLS, an EDSL requires that the developer write their application in a form amenable to constructing a FPGA

implementation, and consequently still requires a knowledge of the platform.

B. Application Overview

Gothandaraman and coworkers[15] have previously ported a VMC application to an FPGA, directing their efforts to potential energy and trial wavefunction evaluation kernels. Through the use of pipelining and fixed point arithmetic, significant improvements in performance were obtained relative to a serial multicore implementation. However, their work was limited to bosonic systems, the trial wavefunction consequently being expressible as a product of functions of pairwise particulate distances, i.e. there is no need to invoke a determinantal form for the trial wavefunction since the system's wavefunction is not subject to the antisymmetry constraints introduced by the Pauli exclusion principle. We have targeted fermionic VMC, and as a result our implementation requires significant departures from previous efforts.

To fully exploit the reconfigurability of an FPGA, we have chosen to write an in-house VMC code so as to not restrict ourselves to the data structures and algorithmic workflows utilised in more sophisticated packages[2, 8, 21]. Rather, we have been able to write the application so as to optimise computation through co-processing with an FPGA. It is worth reiterating that we consider here only a subset of VMC functionality; our implementation is all-electron and Jastrow-free.

We have aspired to optimise our CPU implementation so as to possess a reasonable benchmark. Our VMC code is written in ISO C99, with all data structures written in a "struct of arrays" (SoA) format to facilitate auto-vectorisation and caching through spatial locality of reference. The application is multithreaded using OpenMP. We have attempted to replicate the benefits of code encapsulation offered by object-oriented approaches; a struct is utilised as a means for storing data and function pointers. The Ensemble_t struct, for instance, comprises electron positions, Slater matrices, laplacians, and all other data one may associate with a walker, along with methods for manipulating these data.

Our application foregoes physically motivated data structures to facilitate optimal cache behaviour[22] and facilitate auto-vectorisation by the compiler without the need to resort to intrinsics or external libraries. Such optimisations require a significant effort on the part of the developer, but their implementation can help to ameliorate an application from becoming overly memory-bound. Consider, for instance, a set of n particles, each described by a position vector in $\mathbb{R}^3$. One can conceive of two separate data formats: the contiguous dimensions, i.e. $\{x_1, \ldots, x_n, y_1, \ldots, y_n, z_1, \ldots, z_n\}$; and the physically motivated triplets, $\{x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n\}$. An example of explicit computation of the unique pair-

wise square distances between particles is given in Algorithm 2.

Algorithm 2 Pairwise Distances using Contiguous or Triplet Data Structures

---

```
#define nAtoms 65536
#define nDims 3
static float positions[nAtoms*nDims] ;
static size_t xi, yi, zi = 0 ;
static size_t xj, yj, zj = 0 ;

for( size_t iAt=0 ; iAt<nAtoms ; ++iAt ) {

#ifdef CONTIGUOUS
   xi = iAt ; yi = nAtoms + xi ; zi = nAtoms + yi ;
#else /* TRIPLET */
   xi = iAt * nDims ; yi = xi + 1 ; zi = yi + 1 ;
#endif /* #ifdef CONTIGUOUS */

   for( size_t jAt=iAt+1 ; jAt<nAtoms; ++jAt ) {

#ifdef CONTIGUOUS
      xj = jAt ; yj = nAtoms + xj ; zj = nAtoms + yj;
#else /* TRIPLET */
      xj = jAt * nDims ; yj = xj + 1 ; zj = yj + 1 ;
#endif /* #ifdef CONTIGUOUS */

      float dx = positions[xi] - positions[xj] ;
      float dy = positions[yi] - positions[yj] ;
      float dz = positions[zi] - positions[zj] ;

      float rSq = dx*dx + dy*dy + dz*dz ;

   }
}
```

Since the stride of memory accesses to the positions array in the contiguous format is unity, the compiler is able to automatically vectorise this loop without issue. However, the stride length of three between memory accesses in the triplet format prohibits auto-vectorisation, which is a significant performance penalty for the wide vector registers available on modern processors. The performance benefits from auto-vectorisation are evident from run times; on a machine with SSE2, the contiguous format is over three times faster than the triplet format, as to be expected given the vector register can process four 32-bit numbers simultaneously.

The number of cache misses is also halved for the contiguous format relative to the triplet format. The contiguous format gives significantly improved cache performance since the square distance between particles is computed dimension-at-a-time, i.e. one computes dx,dy,dz in sequence, prior to reduction and computation of the square root. Allowing the compiler to vectorise the code, one finds that the implementation effectively computes dx between iAt and several jAt concurrently. Since the $x-$degrees of freedom are contiguous, a single fetch is adequate to serve the vectorised operation. In contrast, for the triplet storage, one finds that a number of elements in the cache are redundant for the calculation of dx, i.e. the y, z degrees of freedom. As such, the cache does not contain all of the required data to serve the vectorised instruction, resulting in suboptimal caching. Performance can be further improved through the use of techniques such as

strip-mining to improve temporal cache behaviour.

In the following section, we adopt the conventional nomenclature "host" and "device" to denote the CPU and the FPGA, respectively (see Figure 2). The device is connected to the host through PCI express (PCIe), permitting memory transfers to proceed by direct memory access (DMA), i.e. without blocking the CPU. Owing to the fact that VMC is embarrassingly parallel, it should hold that the difference in performance between a host and a host with device is conserved in a multiprocessing environment, with n hosts and n devices. Our study then compares single host against single host with device.

Our FPGA implementation is written in the EDSL provided by Maxeler Technologies, Maxj. Maxj is an extension to java, providing functionalities for dealing with sequences of data which are streamed across an FPGA implementation of the application. Such an implementation is referred to as "dataflow". Maxj is compiled by the MaxCompiler, the resulting executable constructing a register transfer level (RTL) implementation of the application. The RTL is subsequently passed to a synthesis tool (supplied by the chip vendor) to construct the FPGA configuration.

IV. IMPLEMENTATION DETAILS

Throughout, we use the MAIA board produced by Maxeler Technologies, possessing an Altera Stratix V FPGA chip and 48GB of on-board DDR3 DRAM. We are aware that this FPGA is rather out-dated (the Stratix X, recently released by Intel/Altera, has roughly twice the number of resources as the Stratix V used here), but is the best device at our disposal. Our host processor is an Intel Xeon E5-2640 with 6 physical cores, clocked at 2.5GHz. Again, this technology is somewhat out-dated, but has been chosen for a mixture of ease of access and to aid in benchmarking similar semiconductor device fabrications. The molecular system we work with is a lattice of 64 molecules of H_2. We choose the STO-6G basis set for the sake of simplicity, although our described implementation is general enough to utilise an arbitrary basis set.

Evaluating the trial wavefunction and its derivatives is the major computational bottleneck of VMC. For small systems such as Be_2, profiling of our code reveals over 70% of the compute time is localised in the routines which evaluate atomic and molecular orbitals. As the system size increases, the trial wavefunction routines dominate further: upwards of 90% of compute time for the lattice of hydrogen. As such, our target for offloading to the FPGA is obvious.

While the Sherman–Morrison inverse (and determinant) updating routine accounts for a non-negligible portion of runtime, it has been noted that the routine is fairly efficient on CPUs[9] since the inverse matrices fit comfortably in cache. We also note that an improved inverse updating scheme has recently been published[23]. In the future, it may be interesting to incorporate these routines into an FPGA implementation. However, for the purposes of this work, we concentrate our efforts specifically on the trial wavefunction evaluation kernel.

Throughout, we manually set a clock frequency of 100MHz for the synthesised design. We avoid maximisation of the clock frequency with which our application is built as it increases the dimensionality of our design space. For instance, an increase in the clock frequency must be leveraged for other compile-time optimisations, such as depth of pipelines. We do, however, note that preliminary results suggest our design is capable of meeting timing constraints at significantly higher clock frequencies than used here. These results suggest that additional performance is available.

A. Workflow Outline

The algorithmic workflow outlined in Algorithm 1 serves to illustrate the data dependencies of the individual kernels, and the kernels which must be serialised. An electron must be displaced, and the trial wavefunction subject to this displacement evaluated. Should the new trial wavefunction meet the Metropolis criterion, the inverse and laplacian of the trial wavefunction are computed, and the local energy accumulated. Should the trial wavefunction not meet the Metropolis criterion, the electron is reverted to its position prior to displacement. There is no scope for task-level parallelism, rendering it difficult to occupy both host and device simultaneously. Simply offloading the computationally intensive portions of the calculation to the device and blocking the CPU until completion is inefficient from a co-processing perspective. The offload cost must be taken into account when benchmarking against a host-bound application. Any savings made by the co-processor in calculation time must be adjusted for the overhead associated with the offload.

The embarrassingly data-parallel nature of our application can be used to mask the offload cost. By simply splitting our ensemble into two equally-sized ensembles, `Ensemble_A` and `Ensemble_B`, one can be processed by the host while the other is processed by the device. In effect, we artificially create pipeline-parallelism through data-parallelism. However, this enforced task-parallelism comes at the price of additional memory consumption.

Consider momentarily `Ensemble_A` in isolation of `Ensemble_B`. At t_0, it is offloaded to the DFE, where the constituent electrons are displaced, the Slater matrix differences and laplacians are computed, and subsequently sent back to the host. The host is now holds `Ensemble_A` at t_1, where all trial moves have speculatively been accepted. Upon proceeding to the Metropolis step on the host, one finds that upon rejection of a particular move,
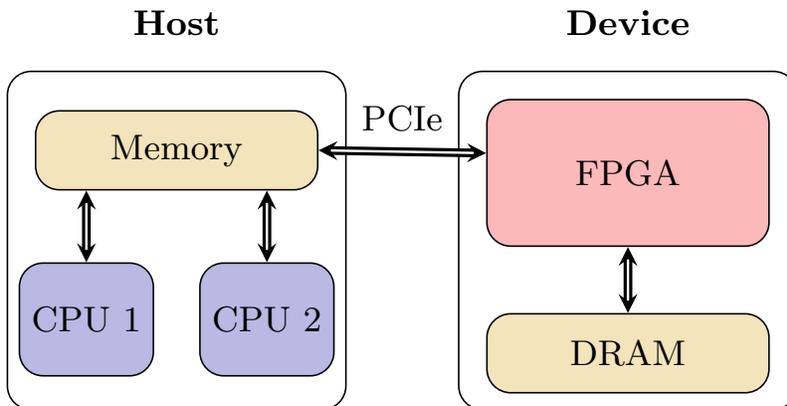
FIG. 2: Schematic of the host-device architecture. A host is a regular multicore CPU, all processing units having access to some bank of main memory. The device is able to interact with the host through PCIe. Memory transactions between host and device can take place between the main memory of the host and the FPGA, which can either utilise the data directly, or route it to device DRAM.

the data associated with the previous configuration has been overwritten, resulting in the requirement that the host recompute the data from t_0.

Now, consider the case where the data associated with `Ensemble_A` is double-buffered. The data at t_0 is stored in both the ensemble's buffers. The double-buffer is offloaded to the device which again speculatively updates `Ensemble_A` to t_1, the data for which is subsequently passed back to the double-buffer. Now, when attempting the Metropolis step, the host has access to data from both t_0 and t_1, allowing either rejection or acceptance of a trial move without the need to recompute data.

Finally, let us reintroduce `Ensemble_B`. Now, while `Ensemble_A` is being processed by the device, `Ensemble_B` can undergo the Metropolis step on the host having previously been speculatively updated on the device. The two ensembles can subsequently alternate being processed by the host and device, resulting in a non-blocking simulation. Furthermore, the host is free to run multithreaded, and the device need never block the host since memory transfers can proceed through DMA (unless the host is undertaking some memory-intensive task, in which case the memory transaction hardware will be overfaced). The number of threads the host uses can be tuned to minimise the amount of idle time and power consumed. An overview of this workflow is given in Figure 3.

B. Dataflow Trial Wavefunction Evaluation

Owing to the enormous computational efforts associated with hardware synthesis, our main aim the construction of a generic dataflow implementation for the trial wavefunction kernel. By generic, we mean that the implementation should be able to process a system of arbitrary complexity (within reason), without the need for resynthesis. Naturally, some constraints must be placed on the FPGA configuration such that it does not simply exhaust all resources on-chip. However, we attempt to minimise these constraints in an attempt to maintain freedom in the design space. In evaluating the trial wavefunction, a number of data parallel regions are amenable to either hardware unrolling or temporal parallelism through the use of stream variables. In constructing a general dataflow solution, our starting point is the identification of a loop which is bounded by some system-independent parameter. The only such loop is given by Equation (8), the reduction over primitives to calculate the value of an atomic orbital. For the vast majority of basis sets, the maximum number of primitives associated with an atomic orbital is 6. As such, we choose to unroll the loop over primitives in hardware. In doing so, an adder tree is also required to perform the reduction over primitives.

From Equation (7), a given atomic orbital $\phi_j(\boldsymbol{r})$ is utilised by all molecular orbital LCAOs. Specifically, a fused multiply-add operation is required to accumulate the atomic orbital multiplied by contraction coefficient onto each molecular orbital accumulation. Since the atomic orbital value is available at the end of the pipeline stage which performs the reduction over primitives, an optimal strategy involves immediate use of the atomic orbital value, permitting the calculation of the next atomic orbital in sequence. We consequently unroll the molecular orbital accumulators in hardware. We undertake a similar accumulation over second derivatives of the molecular orbitals since there is the scope for significant data reuse. It should be recognised that unrolling over molecular orbitals and their second derivatives is equivalent to the accumulation of a row of the Slater matrix and laplacian.

Our choice for temporal unrolling is now enforced: since we accumulate a single row of the Slater matrix and laplacian at a time, a single electron position vector must be presented to the implementation for N_{AO} consecutive kernel ticks. After this period has passed, the row of the
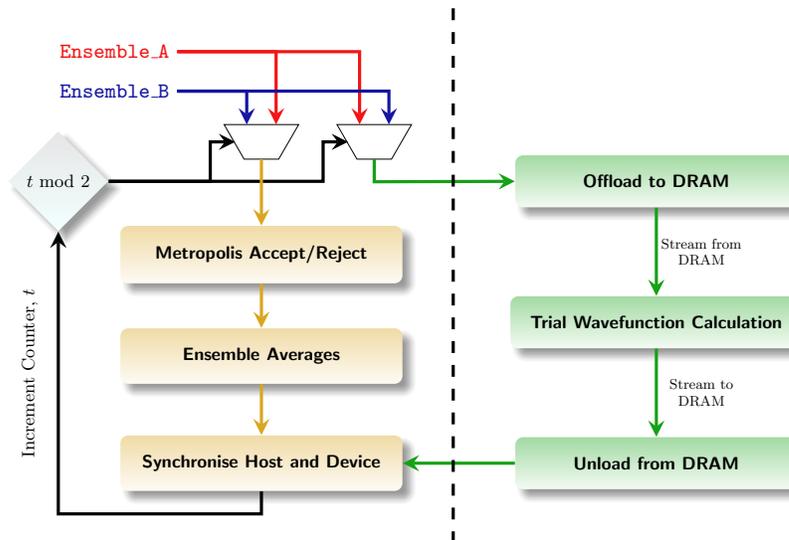
FIG. 3: Non-Blocking implementation of VMC. The dashed line denotes the partition between host (yellow) and device (green) kernels. The destination of the two ensembles alternates between counter ticks, and can be handled by simple ternary logic (represented by multiplexing).

Slater matrix and laplacian will have fully accumulated. These quantities can be offloaded to external memory, the accumulators reset and the next electron streamed through, i.e. the next row of the Slater matrix and laplacian. Since all walkers, and all electrons thereof, must pass through the kernel in this way, we find that our kernel will be required to run for $N_w \times n \times N_{AO}$ ticks. Such counter logic can be realised through three nested counters, over walkers, electrons and atomic orbitals.

A number of quantities are fixed over the course of of the calculation: atomic position vectors, molecular orbital coefficients and primitive parameters. Fast access to these quantities is possible through memory-mapping them to the on-chip ROM of the device from the host. However, we only have access to a counter over all atomic orbitals arising in the LCAO for random access, which could result in significant duplication of data should we construct our memory-mapped ROMs in one-to-one correspondence with the atomic orbital counter. In practice, we utilise a set of decoder ROMs, mapping the atomic orbital counter to the address of quantities corresponding to that atomic orbital in the memory-mapped ROMs. The size of the decoder elements need only accommodate the number of bits to required to represent the total number of atomic orbitals, resulting in significant memory savings. For the system dealt with in our work, the memory savings amount to a factor of five, but this is system-dependent.

The overall schematic for our implementation is given in Figure 4. An electron is streamed from DRAM and subjected to a random displacement. As the displaced electron position vector is required by the host for computing the change in potential energy contribution to the local energy, it is streamed back to device DRAM for offload to the host. The vector and square distance between the displaced electron and an atomic orbital centre is computed, with these quantities being used to compute the values of the primitives associated with the atomic orbital under consideration. A reduction amongst the primitives yields the value of the atomic orbital. The atomic orbital value is subsequently passed onto the molecular orbital accumulators, a fused multiply-add being used to accumulate to the LCAO. Finally, upon complete accumulation, the molecular orbital values are passed to device DRAM.

Our implementation is reliant upon two key kernels which must also be instantiated within our trial wavefunction implementation: a pseudorandom number generator is required for the displacement of electrons, and an exponential function is required for the calculation of primitive values. Our implementation of these two kernels is discussed in detail in Appendices A and B.

C. Numerical Precision

Floating point arithmetic (see Figure 5b) involves several sequentially dependent operations. Addition of floating point numbers, for instance, requires that: the arguments are aligned to the same exponent; the mantissas are summed; the result is normalised (mantissa is shifted and exponent adjusted); and the mantissa is finally rounded to fit within the parameterised bit
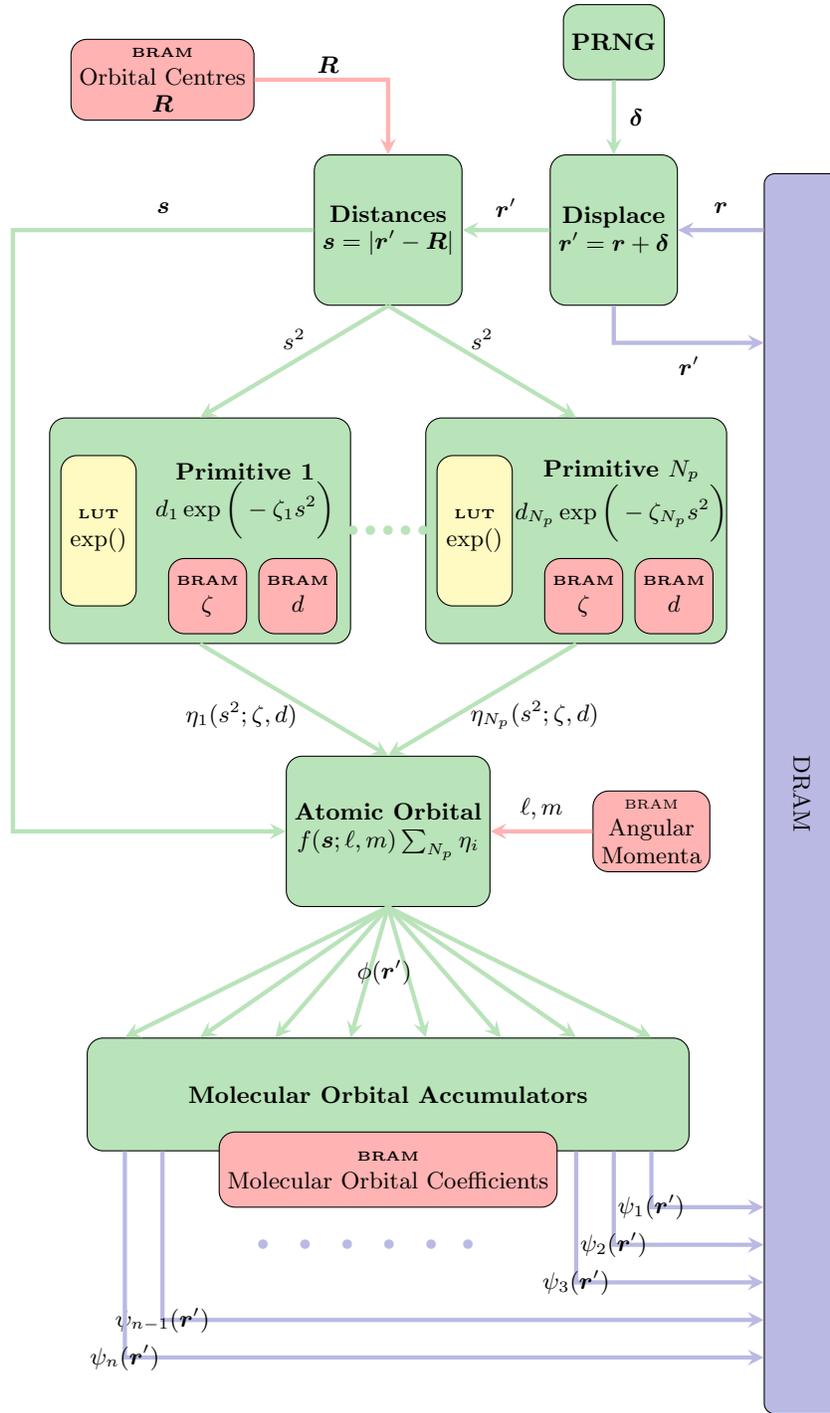
FIG. 4: Wavefunction evaluation dataflow implementation. Red nodes correspond to on-chip memories, yellow nodes to lookup tables. Green nodes are computational kernels, and arrows denote dataflow between elements of the implementation. The blue node represents an on-board addressable DRAM block.

width. Naturally, these operations can be pipelined, floating point arithmetic thereby being suitable for the out-of-order, deeply pipelined architectures of modern computers. Furthermore, the integration of dedicated ICs, such as floating points units (FPUs), into (or sometimes external to) a CPU permits the offloading of these comparatively cumbersome operations to specialised co-processing units without stalling the CPU.

The usage of floating point numerics within an FPGA configuration presents two potential issues: ease of implementation and impact on performance. With regards to the former, a historical obstacle to incor-

porating floating point numerics into FPGA designs has been the requirement that implementations are written with HDL. Maxj permits the definition of a floating point data type through a single invocation of the method `dfeFloat()`, taking both the integer and mantissa widths of the floating point representation as arguments. The data type can subsequently be used much like any other data type, with the implementation details handled by the compile-time synthesis.

Concerning the second issue, the impact on performance of the FPGA design as a result of floating point arithmetic, the associated latency becomes more problematic than for the CPU. Data streams which are to be combined with the result of a particular floating point operation must be buffered on-chip (as must the actual floating point manipulation stages), so as to accommodate the floating point latency. As a result, logical resources are exhausted without contributing to throughput. Some vendors facilitate the usage of floating point arithmetic through the inclusion of digital signal processors (DSPs) within the fabric of the FPGA. However, such resources are limited, and do not remedy the issues associated with buffering other data streams. Indeed, the usage of DSPs places additional constraints on the place and route, inevitably leading to potential difficulties in synthesising a particular configuration.

One particularly appealing solution is to transition from floating to fixed point arithmetic, the latter being equivalent – in terms of resource usage – to integer arithmetic (see Figure 5a). One is consequently faced with a number of choices: the width of the representation; whether the numbers are signed or unsigned (i.e. whether two's-complement arithmetic must be supported); and where the radix point partitions integer from fractional parts, amongst other related decisions. To address each of these choices, we must know both the dynamical range and required precision of the quantities to be manipulated.

There is no native support for fixed point numerics in ISO C99. Nonetheless, it is fairly straightforward to support both fixed point representations and arithmetic through the use of the signed integer data type. However, writing an application which supports fixed point numerics throughout is complicated by a number of issues, particularly when multiple fixed point representations are required:

1. Owing to the lack of support for templated functions and operator overloading by the C99 standard, the host code will be bloated and cumbersome.

2. It is difficult to verify whether an entire application is amenable to fixed point treatment owing to the high-dimensionality of the design space. Supporting fixed point numerics throughout may consequently be an unproductive use of the developer's time should particular operations require floating point treatment.

3. Use of fixed point arithmetic relies on the construction of bespoke mathematical routines. One consequently foregoes the efficiency of the C standard library routines which have benefited from years of optimisation.

4. Should the width of the fixed point data type not be byte-aligned, the effective bandwidth of memory transactions will be suboptimal, leading to poor cache efficiency.

By choosing to use fixed point numerics exclusively in the device code, one is able to circumvent the above obstacles to some degree. One particularly appealing aspect of this choice is the constraint of our fixed point design space to an, admittedly complex, kernel. Furthermore, Maxj possesses native support for fixed point numerics, including exception handling for under-/overflow of the representation along with the inference of representation width from a dynamical range of a quantity. As such, the developer is excused from dealing with low level code optimisation.

However, the question remains of the interface between host and device, i.e. at what point the fixed-to-floating point conversions (and vice versa) take place. The simplest interface involves the offloading of data from the host to the device DRAM in floating point. Upon streaming of data into the DFE, all quantities are cast to and manipulated with an appropriate fixed point representation. Finally, the quantities are cast back to floating point before streaming back to device DRAM. The host is then able to retrieve the data in a natively supported data type.

We find this approach somewhat wasteful. Our kernel fully unrolls the Slater matrix and laplacian streams over the dimensionality of the Slater matrices. Consequently, a large number of casts will be performed concurrently, for both input and output streams. Casting between fixed and floating point representations consumes significant on-chip logic resources. As such, a sizeable portion of on-chip resources will be devoted to casting between numerical representations, an overhead associated with the use of the co-processor. Naturally, such an implementation is best avoided.

A preferable scheme materialises when considering the means by which data is transferred between host and device DRAM. Since the MAIA card resides on the PCIe bridge of a compute node, all data transactions between host and device proceed through PCIe. The PCIe driver is instantiated as an IP core in the fabric of the FPGA, requiring data pass through the FPGA over the course of data transactions. Memory transactions which use PCIe are considered a bottleneck of distributed memory systems. The prudent developer of such applications will consequently minimise memory transactions over the course of computation to obtain an optimised implementation.
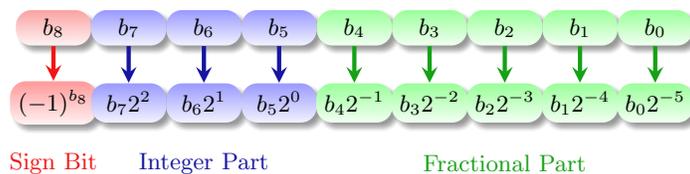
(a) 9-bit fixed point representation. A single "Sign Bit" is reserved to represent the signedness of the number. The "Integer Part" and "Fractional Part" are separated by an implicit radix point. The decimal representation of is given by $(-1)^{b_8} \times (b_7 b_6 b_5.b_4 b_3 b_2 b_1 b_0)_2$
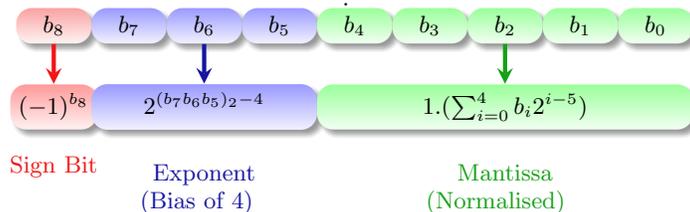
(b)

FIG. 5: Fixed and Floating Point numerical representations using 9 bits (for the sake of demonstration, i.e. without the implication that 9-bit representations are utilised in our work). Note that these forms are inadequate for intermediary stages in arithmetic operations, where guard digits are used to prevent overflow of the representation.

Since data transfer must take place at some point, and the data must pass through the FPGA, it is reasonable to pass the data streams through interface kernels, casting the data as it arrives from the PCIe. A free parameter in our implementation is the width of the stream we cast in the interface kernels. Since the clock frequency of the kernel is known, along with the bandwidth of the PCIe, it is a simple task to evaluate the width of the data stream which saturates the PCIe transfer, and is therefore optimal. For instance, consider a kernel clocked at 100MHz. The bandwidth of a PCIe 2.0 x16 bridge is 8 GB/s. Then, a stream of width 80 bytes saturates the PCIe transfer, equating to 10 double or 20 single precision numbers.

A final consideration is the storage of the fixed point data in the device DRAM should the representation not be byte-aligned. It is difficult to find a generic storage width for arbitrary width fixed point data, since DRAM accesses must be burst-aligned. For example, the MAIA card utilised in this work possesses 6 DDR3 channels, each channel having a width of 64 bits. The DRAM can function with a burst size of 8 or 4, resulting in a burst width of 384 or 192 bytes, respectively. Since the burst width will exceed the PCIe saturated width, data must be buffered in the interface kernel for a number of kernel ticks before offloading the buffer to DRAM. Such a scheme is facilitated by using regular widths for the fixed point storage. As such, we choose a storage width of 64 bits. This choice naturally reduces the effective DRAM latency, but for the sake of ease of implementation, we proceed with this parameterisation.

V. RESULTS

A. Fixed Point

Use of a single fixed point representation throughout our kernel is not a feasible option. A number of variables possess vastly different dynamic ranges. While a single fixed point representation could in principle be found to accommodate all variables, precision of the representation will necessarily be leveraged for flexibility. As such, our strategy is to track the dynamic range of each variable over the course of a simulation and classify the number of fundamentally different dynamic ranges that arise. To narrow our design space, we will constrain each representation to the same width. We are consequently able to eliminate degrees of freedom from the design space that are only amenable to systematic optimisation strategies. In ascertaining an adequate fixed point representation, we must consider three independent points:

1. Whether two's complement is required.

2. The dynamic range of the variables, such that the integer width can be determined.

3. The number of fractional bits required to yield a stable calculation.

The first two points are easily dealt with through analysis. As a further constraint, we concern ourselves specifically with ensembles which have undergone a period of equilibration, thereby further restricting the dynamic range of runtime variables. It is, however, difficult to ascertain the required number of fractional bits without resorting to a full systematic exploration of fractional

widths. Our task is complicated all the more given that in establishing the adequacy of a fractional width, a fully converged VMC calculation is required. Naturally such studies are not feasible when using a device simulator, meaning individual DFEs must be synthesised for each fractional width, which again is prohibitively expensive. As such, we employ a scheme requiring a single Monte Carlo step with the device simulator. The displacement of electrons is deactivated, and the kernel computes the Slater matrices and associated derivatives for the same Monte Carlo samples residing on the host. Upon passing these quantities back to the host, we monitor the ensemble averaged local energy and acceptance rate for these dummy moves. If the computed averaged local energy differs from the original value by less than 5×10^{-4} Hartrees (a.u. hereafter), we deem the implemented fractional bit width adequate. Table I shows such results for a varying number of walkers. We choose this threhold of 5×10^{-4}a.u. since it represents an acceptable standard error in the Monte Carlo estimate for the energy for the calculation to be considered "well-converged". It is clear that a total fixed point width of 38 bits is the appropriate fixed point representation for our calculation.

An interesting optimisation which could be undertaken in the future centres around the work of Ceperley and Dewing[24], where a penalty function is applied to random walks with noisy data. We have no reason to assume that truncating a numerical representation introduces a bias into the resultant calculation, and so it appears that one might be able to utilise a shorter fixed width representation and utilise the reported penalty method to compensate for the resulting imprecision. While the acceptance rate will decrease, one can in principle leverage the inefficiency in the Monte Carlo for potential accelerations deriving from the use of a shorter fixed width representation. We plan to investigate the application of the penalty method to this end in the near future.

B. Performance

The final resource utilisation of our FPGA implementation is given in Table II. While our synthesised design is moderately light on consumption of logic and dedicated arithmetic units, the on-chip memory proves to be a limiting factor to increasing the complexity of our design. Both the LUTs of our exponential units and the on-chip storage of trial wavefunction parameters, particularly the molecular orbital expansion coefficients, are culpable for such high utilisation. However, given the form of our trial wavefunction, avoiding such overheads is difficult. It is possible to force the MaxCompiler to not pipeline sections of the application, thereby reducing on-chip memory consumption, but at the cost of increased difficulty in the design meeting timing constraints. The DSP blocks are effectively only used for multiply-accumulate operations in constructing the LCAOs, hence the low consumption of these particular resources.

Our final results correspond to the overall performance of our application relative to a purely host-bound implementation. We consider a number of ensemble sizes propagated for a fixed number of Monte Carlo steps. In the lower pane of Figure 6, we plot the accelerations relative to the multithreaded host benchmark for a variable number of threads running on the host while the device computes the trial wavefunction. For small ensemble sizes, we observe a reduction in the performance relative to larger ensemble sizes owing to the overhead associated with using the co-processor. However, for ensemble sizes of 256 walkers and above, we see the improvement in performance converge towards an overall acceleration of roughly $30\times$ relative to the multithreaded host implementation. Across all ensemble sizes, we note that the host need only instantiate 3 or 4 threads of execution to tend towards peak performance. An interesting additional result is some metric quantifying the comparative power consumptions of the two calculations. While Maxeler Technologies provide a command line utility to query the power consumption of a board, we are not entirely clear on the resolution or accuracy of this quantity. However, in-house testing on a small board with power readings taken at the power outlet reveals the command line utility is in good agreement with the outlet readings. For the MAIA board, we observed a peak power consumption of 27.6W throughout.

It is, unfortunately, a little more difficult to establish a power consumption of the host. The operating system is ultimately in control of scheduling, so the overhead associated with other general purpose tasks must be accounted for. A further difficulty is what to include in the power consumption for the host. Typically, only thermal design powers (TDP) are reported by chip manufacturers, which are reported to be in poor correspondence with power consumption at peak operation (peak power consumptions equating to roughly $1.5\times$ that of the reported TDPs.[25]). The TDP of the Intel Xeon E5-2640 is reported to be 95W. Furthermore, it is unclear whether the power consumption of RAM chips and other peripherals are to be considered.

Rather than attempt to speculate on such matters, we compose a metric given the limited data at our disposal:

$$E = \frac{P_{\text{benchmark}} \times S}{P_{\text{co-processing}}} \qquad (9)$$

where $P_{\text{benchmark}}$ is the power consumption of the multithreaded host implementation, $P_{\text{co-processing}}$ is the power consumption of the device plus that of the number of threads utilised by the co-processing implementation[26]. S is the speedup of the co-processor relative to the multithreaded host, and E is then the performance-adjusted reduction in power consumption offered by the co-processor. E is plotted in the upper pane of Figure 6. Given that the performance of our application begins to converge to some peak acceleration when the host runs with upwards of three threads, it is unsurprising that the performance-adjusted

N_w	Total Fixed Width						
	30	32	34	36	38	40	Double
128	-65.1254	-67.4323	-68.4111	-69.6823	**-69.6893**	-69.6895	-69.6895
256	-64.2432	-68.1332	-68.0236	-69.7899	**-69.7900**	-69.7901	-69.7901
512	-54.9473	-63.5421	-64.9021	-69.7523	**-69.7892**	-69.7894	-69.7894
1024	-63.5443	-61.5423	-66.3427	-69.6912	**-69.7899**	-69.7899	-69.7899
2048	-67.2313	-68.1998	-69.7712	-69.7652	**-69.7893**	-69.7895	-69.7895

TABLE I: Ensemble-averaged local energy (in a.u.) over multiple numbers of walkers for varying fixed width representations. A sign bit and the number of integer bits have been dictated by the dynamic range of variables. We find that a total width of 38 bits is adequate to reproduce the ensemble-averaged local energy computed in double precision (final column) to within a tolerance of 5×10^{-4} a.u.
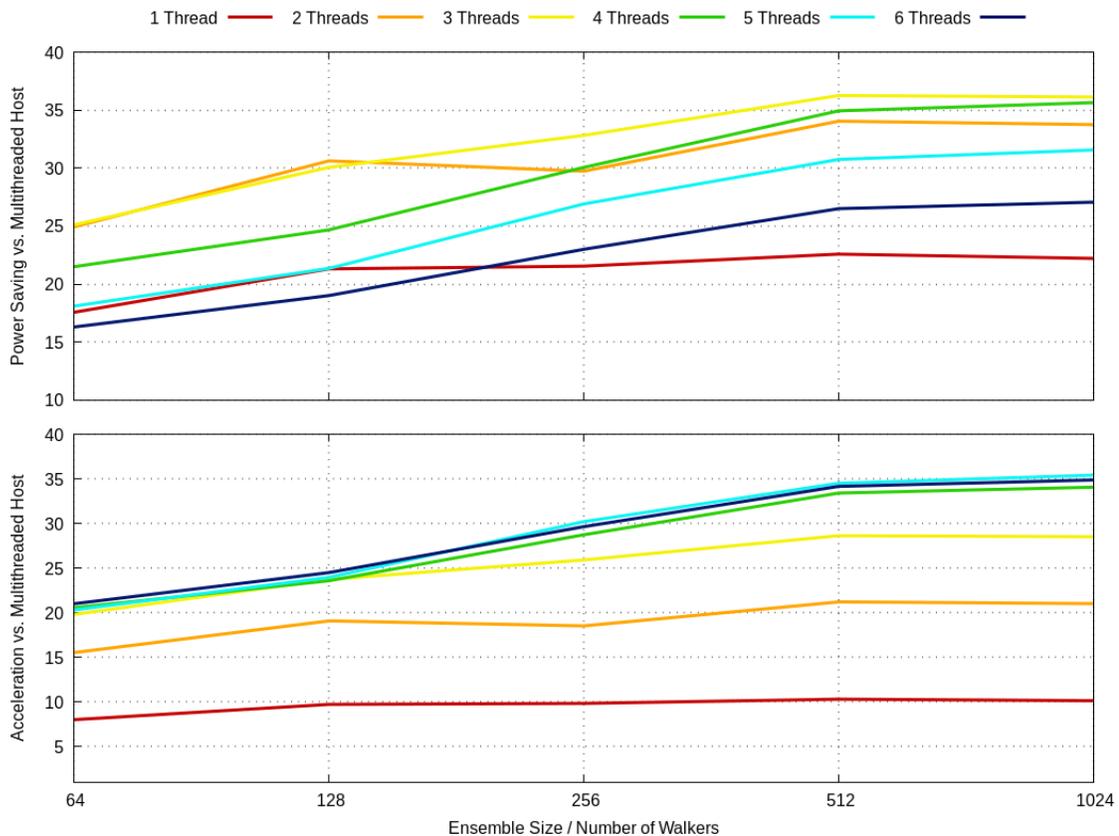

FIG. 6: Acceleration of the host-device implementation for various ensemble sizes relative to a multithreaded host implementation (lower pane). Performance-adjusted reductions in power consumption are also plotted (upper pane).

Component	Number Available	Utilisation
Logic	262400	51.36%
18×18 Multipliers	3926	32.32%
DSP Blocks	1963	36.37%
On-Chip Memory (M20K)	2567	98.64%

TABLE II: Resource utilisation of our final synthesised FPGA configuration for the MAIA.

power consumption is larger for fewer threads than more.

VI. CONCLUSION

We have ported the computationally expensive trial wavefunction evaluation kernel from a VMC application to an FPGA platform. Through co-processing with a multicore host, we have established that our implemen-

tation offers significant benefits in terms of raw compute performance and reduced power consumption. While our VMC is minimal from the perspective of complexity of trial wavefunction, we hope that this work acts to instigate further investigation.

Developer time and on-chip resources remain significantly limiting factors in the use of FPGA co-processing for complex scientific applications. However, these limitations are forecasted to become decreasingly problematic with significant effort being directed towards alleviation. The new Intel/Altera Stratix X, for instance, possesses roughly twice the number of on-chip resources as the Stratix V used in this work.

Work within our group is currently being directed towards adding further complexity to our implementation, including support for Jastrow factors and localised orbitals. We are also looking to support a DMC application, requiring the first derivative of the trial wavefunction, in addition to support for an FPGA-based Sherman–Morrison inverse updating kernel. We hope that such endeavours will aid in the porting of scientific applications to forecasted exascale platforms.

APPENDICES

Appendix A: Pseudorandom Number Generation

The Mersenne Twister (MT) is a pseudo-random number generator (PRNG) whose period is given by $2^z - 1$, z being a Mersenne prime[27]. The most common realisation of the PRNG is parameterised by $z = 19937$, hereafter referred to by MT19937. The period of the resultant pseudorandom sequence is therefore adequately long that repetition will not be observed, even over cosmological periods of time with an output frequency of 1fs^{-1}.

The recurrence of a sequence is a commonly invoked metric for ascertaining the quality of a random number sequence. The MT19937 is k–distributed to 32-bit accuracy, $\forall (1 \leq k \leq 623)$, establishing it as a superior PRNG.

The pseudorandom sequence output by a MT is given by the recurrence relation

$$\boldsymbol{x}_{k+n} = \boldsymbol{x}_{k+m} \oplus \left(\boldsymbol{x}_k^u || \boldsymbol{x}_{k+1}^l\right)\mathcal{A}, \qquad (A1)$$

where $\oplus$ and $||$ denote the bitwise XOR and concatenation, respectively, and n is the recurrence of the sequence. We have utilised vector notation to emphasise that the random numbers are each composed of w–bits; $\boldsymbol{x}^u$ and $\boldsymbol{x}^l$ are then the upper and lower bit segments of $\boldsymbol{x}$. The length of these segments is ascertained by a single parameter, r: $\boldsymbol{x}^u$ corresponds to the upper $w - r$ bits of $\boldsymbol{x}$, and $\boldsymbol{x}^l$ denotes the lower r bits of $\boldsymbol{x}$. Finally, $\mathcal{A}$ is a matrix, termed the "twist-transformation". The twist-transformation matrix is of rational normal form,

$$\mathcal{A} = \begin{bmatrix} 0 & \mathbf{I} \\ a_{w-1} & \boldsymbol{a} \end{bmatrix}, \qquad (A2)$$

where $\mathbf{I} \in \mathbb{Z}^{w \times w}$ is the identity matrix and $\boldsymbol{a}$ is a $w-$bit number, a_{w-1} being the highest-order bit of $\boldsymbol{a}$. This form is most convenient for computation, as one is able to evaluate the vector-matrix product of Equation (A1) through simple bitwise operations, without having to explicitly form and store $\mathcal{A}$,

$$\boldsymbol{x}\mathcal{A} = \begin{cases} \boldsymbol{x} \gg 1 & \text{if} \quad x_0 = 0 \\ (\boldsymbol{x} \gg 1) \oplus \boldsymbol{a} & \text{if} \quad x_0 = 1 \end{cases} \qquad (A3)$$

where $\gg$ is the right-shift operator, and x_0 denotes the lowest-bit of $\boldsymbol{x}$.

To improve the k–distribution of the MT, a "tempering" is used for the output pseudorandom numbers, accomplished through application of a tempering matrix, $\mathcal{T}$. As with the twist-transformation matrix, we are spared from explicitly forming/ storing the matrix $\mathcal{T}$ through enforcing that it satisfy a set of pipelined operations:

$$\boldsymbol{y}_0 \leftarrow \boldsymbol{x} \ \oplus (\boldsymbol{x} \gg u) \qquad (A4)$$

$$\boldsymbol{y}_1 \leftarrow \boldsymbol{y}_0 \oplus ((\boldsymbol{y}_0 \ll s) \ \& \ \boldsymbol{b}) \qquad (A5)$$

$$\boldsymbol{y}_2 \leftarrow \boldsymbol{y}_1 \oplus ((\boldsymbol{y}_1 \ll t) \ \& \ \boldsymbol{c}) \qquad (A6)$$

$$\boldsymbol{y}_3 \leftarrow \boldsymbol{y}_2 \oplus (\boldsymbol{y}_2 \gg l) \qquad (A7)$$

where u, s, t, l are tempering bit shifts, and $\boldsymbol{b}, \boldsymbol{c}$ are tempering bit masks. The ampersand is used in the conventional sense to denote bitwise AND.

The process we have just outlined is most amenable to FPGA implementation. The recurrence relationship of (A1) lends itself to realisation with a "linear feedback shift register" (LFSR), where the output of the register containing an initial sequence of $(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{k-1}$ is simultaneously fed to the back of the LFSR to construct $\boldsymbol{x}_k$, and so on. Furthermore, the pipelined operations outlined in the tempering steps of Equations (A4) - (A7) are suitable for cascaded operations in the fabric of the FPGA. The MT is then a natural candidate as a PRNG for FPGAs, and is our choice for on-chip pseudorandom number generation.

Appendix B: The Exponential Function

Transcendental functions, by definition, cannot be computed exactly in a finite number of algebraic steps. As such, algorithms which implement transcendental functions must employ some approximation, leveraging speed of convergence for an increase in operational complexity. The exponential function pervades quantum chemistry owing to its utility in simplifying the integrals arising in deterministic methodologies. While useful integral properties are an irrelevance for QMC techniques, linear combinations of exponentials are still commonly used to construct atomic orbitals, as stated in Equation (8). As such, in order that our FPGA implementation of the trial wavefunction evaluation routines be as efficient as possible, we have chosen to construct an exponential function which can be queried as rapidly as possible, while consuming minimal on-chip resources.

Historically, the CORDIC algorithm[28] has been utilised on FPGAs for trigonometric and hyperbolic functions. The identity $\cosh(x) + \sinh(x) = \exp(x)$ can subsequently be used to approximate the exponential function. Since CORDIC is composed entirely of adds and shifts, it leads to a significant reduction in complexity relative to polynomial approximations which require explicit multiplication. However, iterative nature of the CORDIC algorithm renders it suboptimal for our purposes – the data dependency between iterations is unsuitable for the dataflow implementation we seek to create.

We have instead utilised a lookup table (LUT) approach[29, 30]. While such implementations are memory-intensive, modern FPGA chips have significant amounts of on-chip RAM (of the order of megabytes), and so the cost of storing a few thousand floating point numbers is fairly inconsequential. In order to utilise a LUT-based approach to approximating the exponential function, one must manipulate the identity

$$\exp(x) = \left(2^{\log_2(e)}\right)^x = 2^{x \log_2(e)}. \tag{B1}$$

We proceed to define $x \log_2(e) = y_i + y_f$, where y_i and y_f are the integer and fractional parts, respectively, of the original argument scaled by $\log_2(e)$. We consequently arrive at the expression

$$\exp(x) = 2^{(y_i + y_f)} = 2^{y_i} \times 2^{y_f}. \tag{B2}$$

A similar partitioning can be undertaken to split y_f down into smaller pieces. Performing this additional partitioning of the fractional part, we can split the approximation down into a series of independent lookups and multiplication of each returned value. The optimal lookup table size is 8K words – partitioning the fractional part into three concurrent lookups and multiplication of the three returned quantities gives a maximum error of 10^{-12} in the approximation to the exponential function. This error is more than adequate given the numerical representations used in this work. Our most accurate fixed point representation, for instance, is Q1.36, which has an associated roundoff error of 10^{-11}.

[1] W. Foulkes, L. Mitas, R. Needs, and G. Rajagopal, Reviews of Modern Physics **73**, 33 (2001).

[2] R. Needs, M. Towler, N. Drummond, and P. L. Ríos, Journal of Physics: Condensed Matter **22**, 023201 (2009).

[3] B. L. Hammond, W. A. Lester, and P. J. Reynolds, *Monte Carlo methods in ab initio quantum chemistry*, Vol. 1 (World Scientific, 1994).

[4] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, IEEE Journal of Solid-State Circuits **9**, 256 (1974).

[5] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous computing with OpenCL 2.0* (Morgan Kaufmann, 2015).

[6] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on* (IEEE, 2011) pp. 365–376.

[7] L. Shulenburger, J. Kim, K. P. Esler, J. McMinis, M. A. Morales, B. K. Clark, and D. M. Ceperley, *Hybrid algorithms in quantum Monte Carlo.*, Tech. Rep. (Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011).

[8] J. Kim, A. T. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley, *et al.*, Journal of Physics: Condensed Matter **30**, 195901 (2018).

[9] K. P. Esler, J. Kim, D. M. Ceperley, and L. Shulenburger, Computing in Science & Engineering **14**, 40 (2012).

[10] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balaji, *et al.*, in *Report of the 2011 Workshop on Exascale Programming Challenges* (2011).

[11] J. Shalf, S. Dosanjh, and J. Morrison, in *International Conference on High Performance Computing for Computational Science* (Springer, 2010) pp. 1–25.

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep **15** (2008).

[13] D. Stroobandt, A. L. Varbanescu, C. B. Ciobanu, M. Al Kadi, A. Brokalakis, G. Charitopoulos, T. Todman, X. Niu, D. Pnevmatikatos, A. Kulkarni, *et al.*, in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2016 11th International Symposium on* (IEEE, 2016) pp. 1–7.

[14] C. B. Ciobanu, A. L. Varbanescu, D. Pnevmatikatos, G. Charitopoulos, X. Niu, W. Luk, M. D. Santambrogio, D. Sciuto, M. Al Kadi, M. Huebner, *et al.*, in *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on* (IEEE, 2015) pp. 339–342.

[15] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, and R. J. Harrison, Parallel Computing **34**, 278 (2008).

[16] V. Kindratenko, I. Ufimtsev, and T. Martínez, Proceedings of the Reconfigurable Systems Summer Institute (2008).

[17] B. Cooper, S. Girdlestone, P. Burovskiy, G. Gaydadjiev, V. Averbukh, P. J. Knowles, and W. Luk, Journal of chemical theory and computation **13**, 5265 (2017).

[18] N. Drummond, M. Towler, and R. Needs, Physical Review B **70**, 235119 (2004).

[19] A. Williamson, R. Q. Hood, and J. Grossman, Physical Review Letters **87**, 246406 (2001).

[20] A. A. Freitas and S. H. Lavington, in *Mining Very Large Databases with Parallel Processing* (Springer, 2000) pp. 61–69.

[21] L. K. Wagner, M. Bajdich, and L. Mitas, Journal of Computational Physics **228**, 3390 (2009).

[22] A. Mathuriya, Y. Luo, R. C. Clay III, A. Benali, L. Shulenburger, and J. Kim, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, 2017) p. 38.

[23] T. McDaniel, E. F. DAzevedo, Y. W. Li, K. Wong, and P. R. Kent, The Journal of Chemical Physics **147**, 174107 (2017).

[24] D. Ceperley and M. Dewing, The Journal of chemical physics **110**, 9812 (1999).

[25] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach* (Elsevier, 2011).

[26] When only a subset of physical cores are utilised in the multithreaded co-processing implementation, the power consumption of each core is given by the TDP divided by the number of physical cores.

[27] M. Matsumoto and T. Nishimura, ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**, 3 (1998).

[28] R. Andraka, in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays* (ACM, 1998) pp. 191–200.

[29] C. C. Doss and R. L. Riley, in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on* (IEEE, 2004) pp. 229–238.

[30] E. Jamro, K. Wiatr, and M. Wielgosz, in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on* (IEEE, 2007) pp. 718–721.