Computing integrals for electron molecule scattering on heterogeneous accelerator systems

Gillan, C. J., & Spence, I. (2021). Computing integrals for electron molecule scattering on heterogeneous accelerator systems. *Concurrency and Computation: Practice and Experience*, *33*(5), [e5984]. https://doi.org/10.1002/cpe.5984

Published in:
Concurrency and Computation: Practice and Experience

Document Version:
Publisher's PDF, also known as Version of record

Received: 8 January 2020 | Revised: 26 May 2020 | Accepted: 22 July 2020
DOI: 10.1002/cpe.5984

WILEY

RESEARCH ARTICLE

Computing integrals for electron molecule scattering on heterogeneous accelerator systems

Charles J. Gillan | Ivor Spence

The School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast, UK

Correspondence
Charles J. Gillan, The School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, 18 Malone Road, Belfast, Northern Ireland BT9 5BN, UK.
Email: c.gillan@qub.ac.uk

Funding information
FP7 Information and Communication Technologies, Grant/Award Number: 610509; Horizon 2020 Framework Programme, Grant/Award Number: 671603

Summary

Using heterogeneous accelerators to obtain high performance for mathematical kernels remains an active research frontier in computational science. The accelerators have compute architectures that are different from the CPUs and in addition have memory spaces independent of the CPU systems to which they are connected. It follows that accelerators require a different approach to writing optimal code than that needed on a multi-CPU system. Taken together these issues have represented a significant barrier to widespread adoption of accelerators for execution with large legacy code bases. OpenCL has emerged as a common programming language with which to implement code that runs across a range of parallel architectures, including multi-core CPUs. This article is a case study on how the instruction-level parallelism offered by field programmable gate arrays (FPGAs) and GPUs through OpenCL can be exploited in molecular physics. The algorithm which we study is the evaluation of tail integrals between Gaussian type basis functions for the R-matrix method, a task that arises in the study of scattering of low energy electrons by molecular targets. The results of our productivity study, which is the first application of OpenCL in this problem domain, show that significant performance can be obtained from both FPGA and graphics processing unit (GPU) accelerators for this application. We discuss suitable transformations unique to each accelerator architecture for the integrals studied and present performance results comparing the FPGA and GPU with execution on Intel multi-core systems.

KEYWORDS
FPGA, GPU, heterogeneous accelerator, integrals, multi-core CPU, OpenCL

1 | INTRODUCTION

The use of heterogeneous accelerators in high-performance numerical computing has increased significantly over the past two decades. The future trend is towards heterogeneous instruction set processors with multi-core CPUs as one component but with custom data paths intermixed. Re-configurable hardware, that is field programmable gate arrays (FPGAs) in addition to graphics processing units (GPUs) are the most common additional compute elements in these new heterogeneous systems. From the perspective of an application developer, one key metric against which any new compute technology is judged is the ease with which it can be programmed (productivity). In scientific and engineering computing there are many large software packages which have been continuously developed since the late 1960s. The ideal would be to take this existing legacy

code and compile it unmodified to run on the combined CPU accelerator environment. Put simply, this is not yet generally possible. Adoption of new programming languages, compiler tools, and system components for accelerators represents a significant barrier to entry for many.

One reason for this situation is that creating a compiler which can automatically generate parallel executable code is a staggeringly difficult task, as seen by the intractability of the problem in the face of the huge resources that have been targeted at it.[1,2] By the time an algorithm is expressed in, for example, Fortran or C, it involves programming features such as indirect addressing, pointers, recursion, indirect function calls, and possibly accesses to global resources. It then becomes essentially impossible for an automated tool to unravel the true nature of the algorithm. It follows that when a piece of legacy code is to be implemented to run on a heterogeneous accelerator platform, it is generally necessary for a human to perform the process of reverse engineering the algorithm and reformulating it for the new environment. When accelerators first entered the domain of HPC, there were different tool kits for each technology. The situation has improved in recent years. On the one hand, the Portland Group provides a Fortran compiler[3] that can target kernels onto GPUs, a product that addresses the needs of the very large base of legacy code developed in the Fortran language. An alternative approach is to port selected kernels from Fortran to OpenCL, via an implementation in C.

We have sought to evaluate this process in this article in respect of one algorithm. We chose to look at the computation of tail integrals in the R-matrix method applied to electron molecule scattering.[4,5] This is an HPC application that has been developed over several decades years in the Fortran language and is today composed many tens of thousands of lines of code. The key contribution of our work is that it is the first implementation of any part of the molecular R-matrix program suite on both an FPGA accelerator or a GPU accelerator and on any platform using OpenCL. Our work quantifies the programmer effort needed to undertake the transformation this large code base in order to enable use of OpenCL. The work also reveals that new code transformations are required to gain maximum performance from an FPGA compared to those needed on a GPU. This article extends our previously published work on the implementation of two-dimensional numerical quadrature for integrals arising the modeling of electron atom scattering.[6,7]

The article begins by presenting in Section 2 some background to the electron scattering problem. Section 3 defines the integrals and presents formulas for their evaluation. The table based algorithm that we developed for the use of accelerators in this problems is described in Section 5. Section 4 presents the programming environment which is used on each accelerator system. Ultimately, this article is about the application of those environments to derive performance from an initial FORTRAN source. The common theme is the porting of this source, but in different ways each time, to each of the three distinct accelerator platforms and programming environments. This means that we treat the accelerators independently in this article; there is currently no one-size-fits-all solution to our porting problem. We discuss related work on the application of accelerators for integral computations in section 7 and this article finishes with a discussion of conclusions from our work and future directions for it in Section 8.

2 | SOLVING THE SCATTERING EQUATIONS

2.1 | The inner and outer regions

The application of the R-matrix method to electron molecule scattering has been described in detail in previous papers[8-10] and we do not repeat the comprehensive details here; we present only the salient details relevant to the study undertaken in this article. Several review articles have also been written and the interested reader is directed to these for a comprehensive description of the method.[5,11]

The R-matrix method solves the nonrelativistic Schrödinger equation for the systems consisting of the impact electron plus target molecular system

$$H\Psi = E\Psi, \tag{1}$$

for each total energy E. Ψ is the unknown wave function which the method seeks to find. H is the Hamiltonian operator composed of kinetic energy and potential energy terms. We assume that the nuclei are fixed in space, an approach known as the Born-Oppenheimer approximation. The H operator therefore transforms as the totally symmetric irreducible representation of the point group to which the nuclear configuration belongs, or stated alternatively it is a rank zero tensor. It follows that Equation (1) must be solved for each irreducible representation of the spatial point group. Given that the nonrelativistic H operator is independent of spin, the quantum mechanical operator for spin commutes with H, so that the calculations must also be repeated for each possible overall spatial and spin quantum number of the system.

The R-matrix method starts by using the Born-Oppenheimer approximation which means that the position of the nuclear framework is fixed in space. The configuration space of electron co-ordinates is partitioned into two regions by surrounding the nuclei with a hyper-sphere, radius a. The inner region is defined as that part of configuration space where $r_i < a$ for all electrons i. The efficiency of the R-matrix method lies in the fact that within the sphere the Ψ is expanded in basis functions, therefore turning equation (1) into a matrix diagonalization problem where the elements of the Hamiltonian matrix are multi-dimensional integrals among products of the basis functions. Most importantly, the diagonalization process is only carried out once per scattering energy per symmetry. The outer region problem is solved for each scattering energy and each symmetry but a simpler approximation is used in the outer region leading to coupled second order ordinary differential equations for the radial part of the functions

representing the scattering electron. The inner and outer solutions are matched at the boundary for each E, a process that generates a matrix known as the R-matrix, from which the method derives its name.

2.2 | Basis functions

Quantum chemistry codes use Gaussian-type functions (GTFs), primarily to benefit from the fact that all the required multi-center integrals over all space can be evaluated in closed form.[12] Furthermore, the double coset method[13] expedites the construction of symmetry adapted integrals. Solving the Hamiltonian problem to compute bound states, which means solving equation (1) for negative values of E, requires integrals over all space. Many computer programs have been developed to perform this task optimally on different computer architectures.[14,15] This body of work has considered the cache efficiency, the number of function calls, loops, and condition branches, streaming SIMD extensions and general-purpose computing on graphics processing.[16] It has been shown that each of these aspects can impact the computation of molecular integrals by as much as an order of magnitude.[17]

The UK molecular R-matrix suite is based on the IBM Alchemy computational chemistry package.[18] Parts of the Molecule-Sweden package[19] were merged into the code base[4] to enable Gaussian type basis functions to be used. This extension allowed polyatomic target molecules to be studied, that is, molecules where the nuclear framework transforms according to the Abelian point group D_{2h} or one of its subgroups. The approach adopted in the UK molecular R-matrix suite[4] is to represent the scattering electron by locating diffuse GTFs on the center of gravity of the molecule, to compute integrals over all space using existing bound state codes and then to subtract the contribution to the integral from the outer region. In this article, we use the code which was added to the R-matrix suite to compute the integrals (known as tail integrals) over diffuse Gaussian function in the outer region and explore the challenges of developing it to use heterogeneous accelerators as well as parallel language features present in the latest version of C++.

2.3 | The role of floating point precision

The absolute value of the integrals, with which this article deals, span a dynamic range of 14 decimal places. It is, therefore, common place to use IEEE double precision for all variables in quantum chemistry integration programs. This is particularly significant when computing relative small tail integrals which are then subtracted from the larger integral values over all space. With the advent of GPUs, some groups have investigated mixed precision single-double precision computations for molecular bound states.[20] This approach is not viable in the case of the R-matrix computations.

This requirement mitigates the performance gain that can be achieved from heterogeneous accelerators. GPUs operate at lower frequencies than CPUs but they typically have many times the number of cores than a CPU chip. However, while all GPUs have single precision units, the number of double precision units varies between micro-architectures. This means that GPUs can in principle perform many times more single precision computations per second than a multi-core CPU but a smaller multiple of double precision calculations. We report results in a later section using a typical desktop GPU, an instance of the NVidia Maxwell 2.0 architecture, on which double precision performance is 1/32 of single precision performance.

FPGA accelerators present a different challenge, essentially offering programmable silicon. In principle, one can implement several floating point units with any number of bits allocated to the exponent and the mantissa. In practice, the OpenCL language currently limits this to the IEEE single and double data types. The issue then becomes whether one can fit the bitstream into the silicon available on the FPGA model. We discuss this later in the article. Furthermore, while the FPGA provides an opportunity to create a multi-core system, the preferred mechanism is to pipeline the computations. This relies on the fact that there is a very high bandwidth network on the FPGA fabric to pass data from one compute element to the next.

3 | COMPUTING THE INTEGRALS

In this section, we present the formulas that are employed to compute the tail integrals which have been explained in the previous section. There are several types of integral in the computation but we focus in this article on the three one electron integrals: overlap, kinetic energy, and nuclear repulsion. We start the section by defining exactly the Gaussian type functions which form the integrand in each case.

3.1 | Gaussian type basis functions

The definition of a primitive Gaussian function with Cartesian powers i,j,k and with exponent α_l centered on the nucleus n located at position (x_n,y_n,z_n) is given at the point (x,y,z) by

$$G_{ijk,n}(x,y,z) = (x-x_n)^i\ (y-y_n)^j\ (z-z_n)^k\ \mathrm{e}^{-\alpha_{ijk} r_n^2}, \tag{2}$$

where the square of the radial distance r is defined by

$$r_n^2 = (x - x_n)^2 + (y - y_n)^2 + (z - z_n)^2. \tag{3}$$

In order to compute the Hamiltonian matrix, one needs to computer overlap, kinetic energy, Bloch, and nuclear repulsion integrals each of which involves products of two primitive Gaussian functions.

In the R-matrix method, the integrals are restricted to a sphere with a finite radius around the center of gravity of the system, this being taken as the origin of co-ordinates. This poses no problem for primitive Gaussian functions located on the nuclear centers because these functions have essentially zero magnitude near the boundary of the sphere meaning that the integrals computed over all space, by a bound state code, can be taken to be equal to the integrals computed over the spherical region. However, in order to model the scattering electron, diffuse primitive Gaussian functions are located at the origin and these do have a finite magnitude close to and beyond the bounding sphere.

It would be extremely challenging to develop a numerical integration method to compute multi-center integrals over GTFs within the R-matrix sphere, radius a. In contrast, it is straightforward to compute such integrals with one radial co-ordinate in the range $[a,\infty]$, hence giving rise the name tail integral. Given that the GTFs are continuous functions, it follows from the properties of integration that the integral over the inner region can be computed as the difference of the integral over all space and the tail integral. Given the limitations of computer arithmetic, computations need to be monitored for the impact of round-off error in this subtraction.

The geometry of the bounding sphere implies that the integrals are easiest to compute when expressed in spherical polar rather than Cartesian co-ordinates. Using the transformation

$$x = r\sin\theta\cos\phi \quad y = r\sin\theta\sin\phi \quad z = r\cos\theta, \tag{4}$$

then defines the diffuse primitive Gaussian functions located on the center of gravity as

$$G_{ijk,n}(x,y,z) = r^{i+j+k}\sin^{i+j}\theta\cos^k\theta\cos^i\phi\sin^j\phi\ \mathrm{e}^{-\alpha_{ijk}r^2}. \tag{5}$$

3.2 | Overlap integrals

The fundamental integral to be evaluated is the three dimensional overlap integral between two primitive diffuse Gaussian, which is shown in Equation (6).

$$I_{\text{overlap}} = \int_a^\infty \int_0^\pi \int_0^{2\pi} G_{ijk,a}(r,\theta,\phi)\ G_{lmn,b}(r,\theta,\phi)\ r^2\ \sin\theta\ \mathrm{d}r\ \mathrm{d}\theta\ \mathrm{d}\phi. \tag{6}$$

Switching to Cartesian co-ordinates equation (6) can be expressed as the product of three distinct one dimensional integrals, as follows:

$$I_{\text{overlap}} = I_{\mathrm{r}}\ I_\theta\ I_\phi, \tag{7}$$

where the one dimensional integrals are

$$I_\phi = \int_0^{2\pi} \cos^i\phi\sin^j\phi\ \mathrm{d}\phi \tag{8}$$

$$I_\theta = \int_0^\pi \sin^{i+j+l+m}\theta\ \cos^{k+n}\theta\ \mathrm{d}\theta \tag{9}$$

$$I_{\mathrm{r}} = \int_a^\infty r^{k+2}\exp^{(\alpha+\beta)r^2}\ \mathrm{d}r. \tag{10}$$

The values for the indices i,j,k shown in these integrals will differ for every pair of primitive Gaussian basis functions. The integral in Equation (8) is zero unless $i,j \in \{0,2,4,\ldots\}$, while the integrals in Equations (9) and (10) can be computed using recursion formulae. Furthermore the integral in Equation (8) can be precomputed for several values of i,j with these values being stored in a lookup table.

3.3 | Kinetic energy integrals

The kinetic energy operator is decomposed into sums over integrals between two primitive diffuse Gaussian functions in the form shown in Equation (11).

$$I_{\mathrm{ke}} = \int_a^{\infty} \int_0^{\pi} \int_0^{2\pi} G_{ijk,a}(r,\theta,\phi)\ \nabla^2\ G_{lmn,b}(r,\theta,\phi)\ r^2\ \sin\theta\ \mathrm{d}r\ \mathrm{d}\theta\ \mathrm{d}\phi. \tag{11}$$

Considering the x-component of the ∇^2

$$\frac{\partial^2 (x^l y^m z^n \mathrm{e}^{-\beta(x^2+y^2+z^2)})}{\partial^2 x}, \tag{12}$$

generates different results depending on the value of i.

$$i = 0: \quad -2\beta \mathrm{e}^{-\beta x^2} + 4\beta^2 x^2 \mathrm{e}^{-\beta x^2} \tag{13}$$

$$i = 1: \quad -6\beta x \mathrm{e}^{-\beta x^2} + 4\beta^2 x^3 \mathrm{e}^{-\beta x^2} \tag{14}$$

$$i = 2, \ldots: \quad i(i-1)x^{i-2}\mathrm{e}^{-\beta x^2} - 2\beta(2i+1)x^i \mathrm{e}^{-\beta x^2} + 4\beta^2 x^{n+2}\mathrm{e}^{-\beta x^2}. \tag{15}$$

Following the same steps for the y and z co-ordinates, it follows that the integral in Equation (11) can be expressed as a summation of integrals in the same form as the overlap integral.

3.4 | Nuclear repulsion integrals

The nuclear repulsion integrals between two primitive diffuse Gaussians have the form shown in Equation (16).

$$I_{\mathrm{nucrep}} = \int_a^{\infty} \int_0^{\pi} \int_0^{2\pi} G_{ijk,a}(r,\theta,\phi) \frac{1}{|\mathbf{r}-\mathbf{R}_a|} G_{lmn,b}(r,\theta,\phi) r^2\ \sin\theta\ \mathrm{d}r\ \mathrm{d}\theta\ \mathrm{d}\phi, \tag{16}$$

where these are defined for every nuclear center in the molecule. The key to evaluating these integrals is to expand the repulsion term as follows

$$\frac{1}{|\mathbf{r}-\mathbf{R}|} = \sum_{lmq} \frac{R^l}{r^{l+1}} S^q_{lm}(\hat{r}) S^q_{lm}(\hat{R}) \quad \text{where} \quad r = |\mathbf{r}|, \qquad R = |\mathbf{R}| \qquad \text{and} \quad r >> R, \tag{17}$$

where the S_{lmq} in Equation (17) are the real spherical harmonics derived from the complex spherical harmonics. A key step is to re-arrange the term on the right-hand side of Equation (17) to

$$\sum_{lmq} \frac{1}{r^{2l+1}} r^l S^q_{lm}(\hat{r}) R^l S^q_{lm}(\hat{R}), \tag{18}$$

because the product of a radial power with a real spherical harmonic can be expressed as

$$r^l S^q_{lm} = \sum_{ijk} c_{ijk} x^i y^j z^j, \tag{19}$$

that is as a linear combination of powers of Cartesian co-ordinates, x,y,z. Using this approach means that the nuclear repulsion integral (16) can be decomposed into a summation of overlap type integrals multiplied by coefficients. It follows that a generic kernel can be developed for the overlap type integrals and then driver functions used for the kinetic energy and nuclear repulsion integrals which make repeated calls to this kernel.

4 | COMPUTING PLATFORMS

We ported existing code from the R-matrix suite to the OpenCL language and then compiled and executed this on different hardware platforms. First, we explain the role of OpenCL and present details of the platforms on which we executed the compiled code.

4.1 | Programming environment: OpenCL

The Open Computing Language, known more commonly as OpenCL, is an open standard maintained by the Khronos Group. It has emerged from the need to provide a common language for implementation of programs that can execute on heterogeneous platforms. This includes CPUs, GPUs, and FPGAs. OpenCL, which is based on C99 and C++11, enables programming these devices with a standard interface for parallel computing using both task and data-based parallelism. A typical approach in OpenCL is to write a kernel which is a procedure in software that can be executed independently concurrently many times, a model that matches to processing 1, 2, or 3D stencils in various applications. There are generally many more instances of the kernel than the processing elements (PEs), the actual hardware on which the kernel executes, and the system schedules blocks of kernels onto the PEs. This model, known as the NDRange approach, is best used for large groups of completely independent parallel work-items that require as few shared variables and as little synchronization as possible. It is possible to execute different kernels at the same time on the same device. This model reflects a host-centric view of accelerators, where communication of data between host and accelerator or between different kernel executing concurrently on the same device write to, and read from, a global memory accessible to all PEs.

While the Intel-Altera implementation of OpenCL implements ND Range feature, the preferred strategy for optimization is to use pipeline parallelism, where iterations of a loop are launched as if they were parallel work-items. Dependencies between iterations are allowed because the hardware generated by the toolkit ensures that a value computed in one iteration is available before it is read by a subsequent iteration. The Altera off-line compiler (AOC) reads an OpenCL code and implements pipelines for the computations inside for loops. Since FPGAs have a large number of registers, pipelines with hundreds of stages can be created. Nested loops and data dependencies within loops need careful consideration, otherwise loop-stalls can be created which impact negatively on performance.

4.2 | FPGA accelerator

The computing platform on which we executed our code consists of an Intel server equipped with a Nallatech P385-A7 card. This half-height, half-length card is inserted into a PCI Express slot and operates as an 8-lane PCI Express Gen2 peripheral. The card is capable, according to manufacturer documentation, of operating at Gen3 speeds but the standard OpenCL SDK and Nallatech board support package limit its operation to Gen2. The compute element on board is an Altera Stratix V GX A7 FPGA surrounded by two 4 GB banks of DDR3 SDRAM. We prepared bitstreams for the system using the Altera SDK for OpenCL, 64-Bit version 15.1.2 Build 193.[21]

The Intel server hosting the Nallatech 385-A7 card has one Xeon 5530 processor with four physical cores operating at 2.4 GHz and each with 8 MB of cache memory. The system is equipped with 24 GB of physical memory. The system tuns the CentOS operating system, build 2.6.32 – 642.4.2.*el6.x*86$_6$4 and code executing on the server ic compiled using version 4.4.7 of the GNU compiler suite.

4.3 | GPU accelerator

An NVidia GPU is composed of independent stream multiprocessors (SM) which implement a single instruction multiple data (SIMD) style of processing. Each SM has several scalar processors and one instruction unit. The same instruction in executed concurrently on each scalar processor but each processor operates on different data. A kernel code is instantiated once in each thread and the set of all threads are partitioned into blocks. Each SM can process several blocks concurrently, but all the threads in a given block are guaranteed to be executed on a single SM. To gain maximum performance from the GPU, it is customary to have the number of threads much larger than the total number of scalar processing units, and all the threads are executed using time slicing. All blocks are split into groups of thirty-two threads, an entity which is defined as a warp. The thread scheduler maintains load balancing and maximizes overall performance by switching between the active warps.

The micro-architecture of the NVidia GPU product line has evolved continuously since the product was introduced. In our work, we have used the Tesla M60 product has two GM204-400 type GPUs on board. The GM204-400 is an instance of the second generation of the Maxwell micro-architecture however retaining the same CUDA programming model, but with enhancements, as in the older Fermi and Kepler versions. The GM204 supports CUDA programming model 5.2.The GM204 are arranged as four graphics processing clusters each of which has four streaming blocks. In turn, each block has four logic units including 32 cores for 32 bit floating point (single precision, FP32) arithmetic and one core for 64 bit floating point (double precision, FP64). In total then, one instance of the GM-204 has 2048 FP32 compute cores and 64 FP64 compute cores. The base clock rate is 1126 MHz. On the M60 board each GM204 has 8 GBytes of GDDR5 global memory. The OpenCL construct known as NDRange maps directly onto the model of SMs and thread warps.

4.4 | Intel multi-core CPU

In order to compare our work on the accelerator platforms, we executed the code on multi-core Intel computers based on the core i7 CPU chip operating at 2.3 GHz. This is an instance of the Haswell micro-architecture and supports SEE and AVX instructions as well as fused-multiply add instructions. We used one system with one core i7 chip installed with the Windows 7 operating system and the Intel OpenCL toolkit. For the work reported in Table 6, using the C++ std::async functionality, we used a server with 36 physical cores Intel i7 CPUs running version 14 of the UBuntu operating system version 14 with the g++ compiler version 5.4.

5 | A TABLE DRIVEN ALGORITHM TO ENABLE PARALLELISM ON ACCELERATORS

The previous section explains how mathematical form of the various tail integrals arising in the study of electron-molecule scattering with Gaussian type functions can be decomposed, in general, to sums of overlap integrals of the type shown in Equation (6). In the initial sequential version of the code, a generic kernel was implemented for the overlap integrals and then this was called repeatedly within loops over pairs of functions. Figure 1 illustrates this relationship schematically. There are two distinct aspects to introducing parallelism to this computation. On the one hand, one can consider the coarse grain parallelism offered by the fact that each integral is independent of other integrals and we consider this in the next part of this section. On the other hand, one can use fine grain parallelism to optimize the computation within one integral. We present details of this at the end of this section.

5.1 | Identifying parallelism

This loop based approach originally designed for single CPU operation and in limited places exploited the fine grain parallelism offered by SSE units on Intel chips. The aim was to compute each integral, one after another, as efficiently as possible in a loop. Given that the computation of each integral is independent, we redesigned the code completely to be able to use multiple CPUs and/or accelerators to compute integrals in parallel. The key step to enable this was to define the overall algorithm to employ a table driven approach in which we separate the processing logic from the call to the kernel function. This can be expressed in pseudo-code as follows:

1. Scan the set of all basis functions and determine maximum indices *i*,*j*,*k* occurring in Equation (2) and dynamically allocate storage for a table containing all integrals. This table of integrals becomes the primary data structure that enables parallelism in the new code.
2. Using the looping structure from the original code, processing all possible combinations of basis functions that generate non-zero integrals for one or more of the three types of integrals defined in Equations (6), (11), and (16). As explained previously, each kinetic energy and nuclear repulsion integral is expressed as a sum over overlap integrals. This step depends on the point group to which the nuclear framework belongs. In our code, we are limited to the Abelian point group D_{2h} ad its sub-groups. In the case of nuclear repulsion integrals, this means expanding the repulsion term (17) and giving multiple overlap type integrals. For each non-zero integral identified, enter index and exponent details into one row of the integrals table, which is sufficient information with which to compute the integral. The format of the table is illustrated in Figure 2 where data from one of the basis sets reported in Section 6.3 is used. Populating the rows of the table to define the complete set of required integrals requires only integer arithmetic, principally on the Cartesian powers of the primitive Gaussian functions, as well as logic operations.

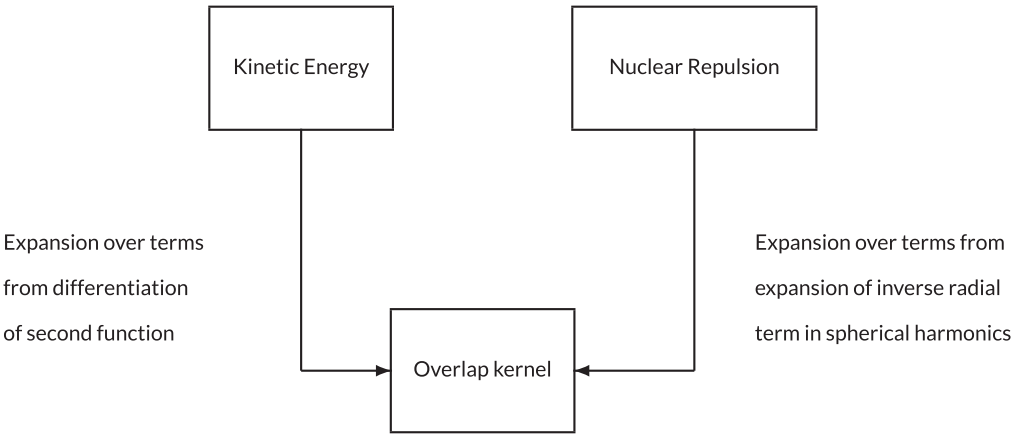

FIGURE 1 Relationship between the different types of integrals and the generic overlap kernel function

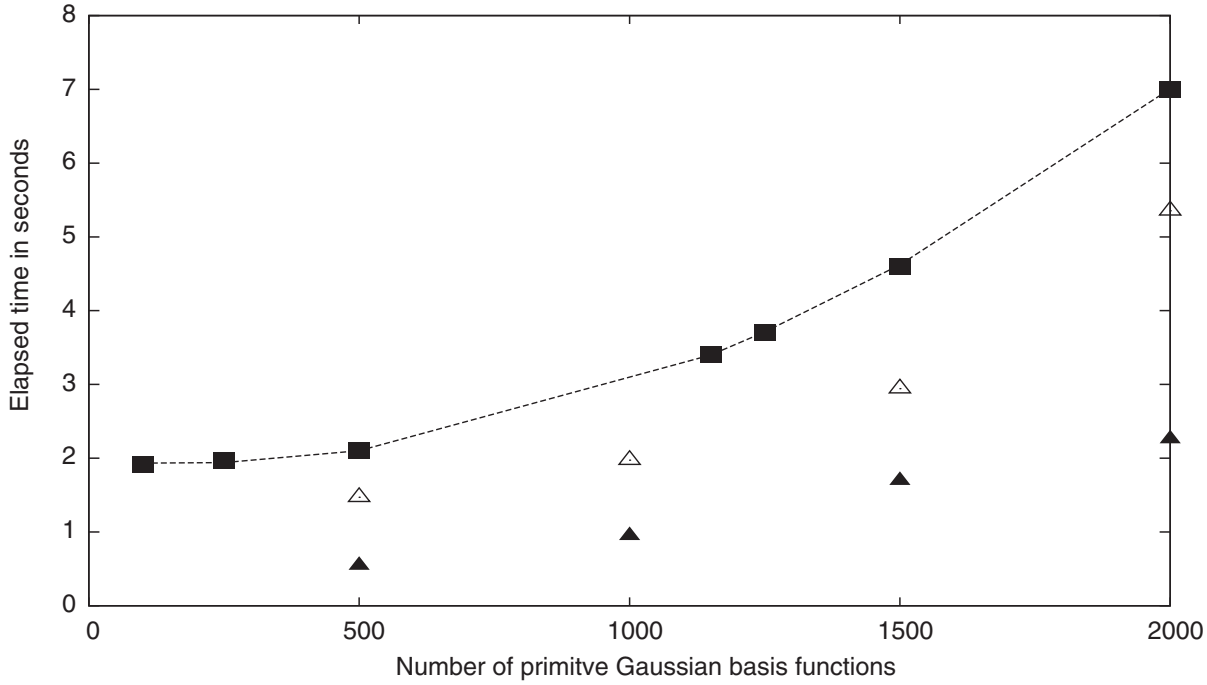

FIGURE 2 The x-axis shows the number of diffuse Gaussian functions used in each calculation, while the y-axis denotes time taken to compute all one electron integrals. ■ points show the total compute time for execution on the Nallatech 385 FPGA accelerator in double precision. The solid line is a quadratic curve fitted to the data; it is discussed further in the text. △ presents the results for execution on Intel quad core i7 system and ▲ shows the results for execution on the NVidia Tesla GPU described in the text

3. When the table of integrals is fully populated with index and exponent details, the integral corresponding to each row can be computed independently and the corresponding value entered into the column of the integrals table. We used different programming approaches on the CPU, GPU, and FPGA to leverage the parallelism offered by table structure.
4. The final step is to process the integrals table on the host to perform sum reductions needed to compute the kinetic or nuclear repulsion integrals.

5.1.1 | Independent tasks: CPU (OpenCL and C++) and GPU

The integrals table on the main CPU is best implemented using a struct of arrays (SoA) approach. in fact we use a C++ std:vector for each column because this is guaranteed to allocate contiguous memory for the elements of the vector. This facilitates transfer to the accelerator global memory, which we describe later, by simply copying the whole column to a pointer on the accelerator. This represents a relatively low cost compared to the computations therefore. Table 1 shows the maximum data speeds for the PCIe buses which connect to the peripherals. The alternative approach of using a defined struct for each row would present the potential problem that the memory pattern used by the compiler for the accelerator may not match that used on the host thereby making data copy operations alone impossible. To further optimize storage, we pack three integer indices into one 32 bit value using 10 bits per index. This exceeds any practical index value that arises in the sets of basis functions in common use today.

On a multi-CPU shared memory system, the integrals can be computed by creating a number of asynchronous tasks, each with access to the integrals table. Each task has an index which can be used as an initial row of the table, the first integral that it computes. For example, the task with index 0 can compute the first entry in the integral table. If the number of asynchronous tasks in operation is 4, this task with index 0 can then compute rows 4,8,12, … in the integrals table until it reaches the end of the table. This approach is illustrated using C++ language features by the following code excerpt from our work.

```
std::vector<std::future<void>> futures\_vec;
int const ntasks = 4; // Can be tuned to available CPUs
for(int itask = 0; itask < ntasks; ++itask)
   {
```

```
    futures\_vec.push\_back(std::async(std::launch::async,          // Launch policy
                                        gtolap\_cpu,        // Kernel
                                        itask,             // Index of task
                                        ntasks,
                                        &(ijkvec[0]),    // Columns of the
                                        &(alphavec[0]),  // table of
                                        &(lmnvec[0]),    // integrals
                                        &(betavec[0]),
                                        &(phi\_ints\_vec[0]),
                                        nrows\_tot,
                                        maxl4,
                                        0,
                                        radius,
                                        &(xintvec[0])    // Value of integral
                                    )
                           );
    }

// Wait for tasks to complete

for(auto &e : futures\_vec)
    {
     e.get();
    }
```

Listing 1: List 1

This is readily enabled by instantiating a vector of instances of the std::future class template. Each instance of std::future wraps the kernel function, gtolap_cpu(), which processes the integrals table in the way discussed in the previous paragraph (Table 2). We mapped this to the multi-core CPU using the std::async feature of C++11 with the policy std::launch::async. This means that the runtime continues to launch tasks in separate threads as long as there are unused CPUs available on the system. With modularized code, the use of the asynchronous features in C++ is a relatively straightforward process.

A similar approach can be achieved on the CPU using OpenCL or equivalently on the GPU. The clEnqueueNDRangeKernel() method is used to launch multiple instances of the integration kernel on the platform, a one dimensional structure. Then each kernel determines its thread index and

TABLE 1 Details of the platforms and OpenCL versions used

Platform/ Clock speed	Operating Power	OpenCL Version	Data bus to accelerator
Nallatech P385-A7	25W	Intel aoc compiler version 16.1.0 Build 196	PCIe 3.0 x8
205 MHz			(7.8 GB/s)
NVidia Tesla M60	300W	NVidia Cuda, release 10.1, V10.1.168	PCIe 3.0 x16
1178 MHz			(15.7 GB/s)
Intel core i7-4712HQ	165W	Intel OpenCL for CPU 16.1.0, packver 5.2.10063	
2300 MHz			
Intel Xeon E5-2695 v4	120W	GNU g++ v.5.4.0	
2100GHz	(18 CPUs)		

TABLE 2 Format of the data table built on the host CPU and shipped to accelerator memory

IGauss	JGauss	LMQ	i	j	k	α	l	m	q	β	λ	Coefficient	Integral
Three distinct overlap integrals …													
70	70	0	0	0	0	0.1081080	0	0	0	0.1081080	0	1.0000000	0.0000001
71	70	0	0	0	0	0.0950950	0	0	0	0.1081080	0	1.0000000	0.0000005
71	71	0	0	0	0	0.0950950	0	0	0	0.0950950	0	1.0000000	0.0000019
Kinetic Energy integral between function 84 and 96 expanded as overlap integrals													
84	96	0	0	0	1	0.0950950	0	0	3	0.0383360	0	0.0058786	1.8264436
84	97	0	0	0	1	0.0950950	1	0	0	0.0291800	0	-0.1750800	0.1139793
84	97	0	0	0	1	0.0950950	3	0	0	0.0291800	0	0.0034059	3.1054991
84	97	0	0	0	1	0.0950950	1	0	0	0.0291800	0	-0.0583600	0.1139793
84	97	0	0	0	1	0.0950950	1	2	0	0.0291800	0	0.0034059	2.3291243
84	97	0	0	0	1	0.0950950	1	0	0	0.0291800	0	-0.0583600	0.1139793
84	97	0	0	0	1	0.0950950	1	0	2	0.0291800	0	0.0034059	6.2109000

Note: Each row represents one overlap integral to be computed. Only selected rows are shown from an actual run.

first computes the integral at this row of the integrals table. It then moves to row (index + number of threads) and computes this integrals etc. The code is implemented in the following way:

```
int const ntotalX  = get\_global\_size(0); // Total threads launched

int const iglobalX = get\_global\_id(0);   // ID for this thread

...

// nrows is the length of the integrals table

for(int irow=iglobalX; irow<nrows; irow+=ntotalX)
   {
    // Compute the integral at this row using sum reduction
    // formulae for the radial and theta parts
    // Read the integral for the phi part from global memory
    ....
   }
```

Listing 2: List 2

5.2 | Pre-computed integrals over the phi co-ordinate

The integrals over the ϕ co-ordinate can be expressed analytically and they are reused several times throughout the computation of the full set of integrals. This means that it is computationally efficient to compute a table of all required integrals at the start of the execution and to reuse it continuously. The dimension of this is determined by summing the powers *i*,*j*,*k* for each basis function and finding the maximum value, maxl, over the whole set of functions. The number of integrals over the ϕ co-ordinate is then 2 maxl $+1)^2$. The table is computed once and shipped to the global memory of each accelerator as a one off cost. It consumes only a few MB at most and is significantly smaller than the table defining the full

set of integrals. The integrals over the θ and ϕ co-ordinates can also be expressed analytically and coded as sum reductions. There is no numerical quadrature required.

5.2.1 | Pipelining for parallelism on the FPGA

In the case of the FPGA, the preferred mechanism for parallelism is that the rows of the table are processed in a pipeline in contrast to the situation on the GPU and on the multi-core CPU. The key to achieving good performance from the FPGA implementation is to aim to have one loop iteration launched every clock cycle, technique. Loop carried dependencies can prevent this causing launched to be delayed by several clock cycles. The AOC compiler provided by Intel/Altera reports for each loop, when the -g flag is set,

- operations that contribute the largest delay to the computation of the loop-carried dependency;
- the launch frequency of a new loop iteration.

We therefore investigated the optimization reports from the compiler to determine where the dependencies were located. The following shows an extract from the optimization report for the first OpenCL version of our code. We have added extra information to this to relate the report to the loops implementing the formulas presented above.

```
================================================================
|    *** Optimization Report ***
================================================================
| Kernel: gtolap
================================================================
| Loop for.body
|   Pipelined execution inferred.
|   Successive iterations launched every 2 cycles due to:
|       Pipeline structure

    Note: This the loop over all rows of the data table
          passed from the host CPU.

----------------------------------------------------------------
| Loop for.body26
|   Pipelined execution inferred.
|   Successive iterations launched every 8 cycles due to:
|       Data dependency on variable angpart1
|       Largest Critical Path Contributor:
|           96

    Note: This is the loop for the computation of the integral
          over the theta co-ordinate.

----------------------------------------------------------------
| Loop for.body73
|   Pipelined execution inferred.
|   Successive iterations launched every 19 cycles due to:
|       Data dependency on variable tail
|       Largest Critical Path Contributors:
```

```
|           35\%: Fadd Operation
|           32\%: Fmul Operation
|           31\%: Fmul Operation

    Note: this is loop for the integral over the radial
          co-ordinate.

================================================================
```

Listing 3: List 3

The loop that computes the integral over the θ co-ordinate is identified above to add a latency of 8 clock cycles, while the loop which computes the integral over the radial co-ordinate reports a latency of 19 clock cycles. In both cases, these loops are written using an accumulator variable which holds, by the end of all iterations, the value of the relevant integral.

The Programming Practices Guide suggests several strategies to remove this kind of loop carried dependency. We investigated the use of shift registers, a technique well-known to VHDL programmers but not generally suited to CPU implementations. Applying this technique means that instead of adding values to a single accumulator, within a loop, an array known as a shift register is first established and all its elements are set to zero. Then at each iteration of the loop, the following additional operations are performed with the shift register:

- each newly computed value from the iteration is added to the element at the base of the array and the result placed at the top of the array;
- all elements in the array are pushed down one location, step which overwrites the base element but duplicates the element at the top of the array.

After the last iteration of the loop is complete, the elements in the shift register are summed, excluding the value at the top, to give the final result. This process can be implemented very efficiently on an FPGA and are widely used in streaming data applications such as network packet processing, digital signal processing, and image processing. The reason that shift registers are optimal on FPGAs stems from the fact that there is no complex address decoding required and that data movement can be enabled via dedicated connections between logic blocks in the fabric. Unlike the CPU implementation, there is no shared data bus. The integral over the θ co-ordinate can be expressed analytically as a sum over terms. In the CPU code, this is implemented in the following loop structure, where the label REAL_t is replaced by float or double as required when compiling and where angpart1 is the summation computed.

```
REAL\_t  coef     = 1.0e+00;
REAL\_t  angpart1 = 0.0e+00;

int const ij2 = ijlm >> 1; // Division by 2

for(int ij = 0; ij <= ij2; ij++)
  {
          int const itemp = kplusn + ij +ij +1;

          REAL\_t const fdenominator = (REAL\_t) itemp;

          REAL\_t const term2 = coef/fdenominator;

          angpart1 = angpart1 + term2;

          int const itop    = ij2 - ij;

          int const ibottom = ij + 1;
```

```
            REAL\_t const ftop    = (REAL\_t) itop;

            REAL\_t const fbottom = (REAL\_t) ibottom;
                 REAL\_t const frac    = ftop/fbottom;

            coef = -coef * frac;
        }
```

Listing 4: List 4

Optimization analysis from the FPGA compiler reported an 8 cycle issue latency when using the float data type, for this loop. Following the Altera Best Practices Guide for summation loops, we transformed the code to use a shift register thereby removing the cycle latency. The transformed code is:

```
#define ANGPART1\_CYCLES 8

        REAL\_t  angpart1\_shift\_reg[ANGPART1\_CYCLES+1];

        for(int iang = 0; iang < ANGPART1\_CYCLES + 1; iang++)
           {
            angpart1\_shift\_reg[iang] = 0.0e+00;
           }

        //

        REAL\_t  coef     = 1.0e+00;
        REAL\_t  angpart1 = 0.0e+00;

        int const ij2 = ijlm >> 1; // Division by 2

        for(int ij = 0; ij <= ij2; ij++)
           {
            int const itemp = kplusn + ij +ij +1;

            REAL\_t const fdenominator = (REAL\_t) itemp;

            REAL\_t const term2 = coef/fdenominator;

            //
            //.... Load this element into into the end
            //     of the shift register
            //
            //     If ij > ANGPART1\_CYCLES, add to
            //     angpart2\_shift\_reg[0] to preserve values
            //
```

```
              angpart1\_shift\_reg[ANGPART1\_CYCLES] = angpart1\_shift\_reg[0] + term2;

              //
              //.... Shift the data in the register
              //
#pragma unroll
              for(int jang = 0; jang < ANGPART1\_CYCLES; ++jang)
                {
                 angpart1\_shift\_reg[jang] = angpart1\_shift\_reg[jang + 1];
                }

              //
              //.... Update the coefficient for the next iteration
              //

              int const itop    = ij2 - ij;

              int const ibottom = ij + 1;

              REAL\_t const ftop    = (REAL\_t) itop;

              REAL\_t const fbottom = (REAL\_t) ibottom;

              REAL\_t const frac    = ftop/fbottom;

              coef = -coef * frac;
             }

           //
           //---- Sum every element of shift register
           //

           REAL\_t angpart1\_shift\_reg\_sum = 0.0e+00;

#pragma unroll
          for(int i = 0; i < ANGPART1\_CYCLES; ++i)
             {
              angpart1\_shift\_reg\_sum += angpart1\_shift\_reg[i];
             }

          angpart1 = 2.0e+00 * angpart1\_shift\_reg\_sum;
```

Listing 5: List 5

This optimization is specific to the FPGA architecture. While the code works on CPU and GPU, it performs poorly on these architectures.

6 | RESULTS

We begin this section by discussing the resource consumption for different implementation on the FPGA. As we have pointed out earlier, only the double precision implementation can produce values that are suitable for use in modeling computations. However, we report some aspects of the implementation of single precision computations to illustrate the resource consumption on the FPGA. Computations reported on for the CPU and GPU are only using double precision. At the end of section, we turn our attention to performance gained from our implementations.

6.1 | Resource usage

Table 3 reports on the resources used on the FPGA by the bit stream prepared for the kernels. the table compares the case of IEEE floating point and double implementations. The table shows that moving from float to double significantly increases the use of DSP blocks increasing from 37% to 85%. The impact on other resources is less significant. The time taken to prepare a bitstream for the FPGA for each case is presented in the final column of the table.

6.2 | Data transfer to/from the accelerator

We have explained earlier in the article that the algorithm is centered on a table of integrals. This is transferred to the accelerator before the launch of the compute kernels and one column, the computed values, is retrieved to the host when the execution of all kernels is completed. Table 4 reports the aggregate transfer times moving the integrals table between the accelerators for different size of problem. These figures can be compared to the row labeled *Execution times* in Table 5. The aggregate data transfer times are generally less than 10% of the total execution time.

6.3 | Performance analysis

We performed several calculations for the case of the nitrogen molecule with the nuclei fixed in its equilibrium configuration. We defined different basis sets of diffuse Gaussian functions located at the center of the module. We initially used a basis set from previous work and then extended this to include higher angular momentum terms. This extension generates more basis functions and therefore more integrals so that we can examine the scalability of the computation. Table 5 shows the number of non-zero integrals computed for various sizes of the basis sets. One of the limiting factors in our approach is to determine whether the table of integrals, which has been described above and an example presented in Table 2 fits within the global memory space of the accelerator. The table contains a row showing the number of bytes required for each size of basis set. The

TABLE 3 Comparison of resource usage on the FPGA on the Nallatech 385 accelerator card for both float and double precision data types

Data type	Logic utilization	Dedicated registers	Memory logic	DSP blocks	Build time (hours)	Comments
float	27%	12%	25%	37%	0.5	No optimizations
float	31%	14%	26%	37%	0.75	Shift reg for θ integral and unrolled coefficients
double	38%	17%	28%	85%	4.0	No optimizations
double	40%	18%	29%	85%	0.0	Shift reg - θ integral

TABLE 4 Aggregate data transfer times (in seconds) transferring the integrals table to/from the GPU and to the FPGA accelerators discussed in the text

Rows in table (integrals)	GPU	FPGA
3,453,704	0.040	0.070
22,178,502	0.190	0.320
39,117,912	0.262	0.416
71,860,567	0.381	0.578

TABLE 5 Details of the number of kernel executions per type of integral between basis functions and size of global memory required to store the data table

	Number of basis functions			
Type of integral	500	1000	1500	2000
Overlap	21 853	117 666	179 091	299 116
Kinetic energy	1 296 880	8 631 360	15 371 070	28 405 740
Nuclear repulsion	2 134 971	13 429 476	23 567 751	43 155 711
Total integrals	3 453 704	22 178 502	39 117 912	71 860 567
Table size MB	151	975	1721	3161
Execution times (s):				
Nallatech 385 (FPGA)	2.167	3.106	4.619	7.085
NVidia Tesla M60 (GPU)	0.554	0.947	1.695	2.259
Intel i7 8x2.3 Ghz (CPUs)	1.471	1.973	2.942	5.353

TABLE 6 Elapsed time in seconds for computation of the integrals using different numbers of asynchronous tasks on a system with 36 physical CPUs

Number	Number of basis functions			
of tasks	**200**	**500**	**1000**	**2000**
2	1.25	1.28	6.79	37.31
4	0.72	3.63	9.51	18.99
8	0.38	2.19	5.22	10.50
16	0.25	1.16	3.05	6.01
32	0.26	0.67	1.78	3.41

Note: No other users were on the system during the execution of these tests.

TABLE 7 Comparison of timings for-loop versus two asynchronous tasks implemented in C++

Number of basis	Elapsed time in seconds	
functions	**for-loop**	**two tasks**
200	0.24	1.25
500	1.34	1.28
1000	9.72	6.79
2000	43.91	37.31

Nallatech 385 board has two banks of 4 GB DDR3 SDRAM. In our code, we access only one of these, limiting the size of the global table to less than 4 GB. In double precision mode, each row of the table requires $44bytes$ to hold values therefore giving an absolute limit of around $90M$ rows, that is, $90M$ integrals to be computed. As Table 2 shows this limit is reached with over 2000 basis functions. This limit could be overcome by dividing a larger table into sub-units and computing these in a loop. Figure 2 shows the time taken to compute the integrals for different number of basis functions. This figure includes, for each case, construction of the table on the host, the time to move the table to and from the FPGA, and the time spent on the FPGA computing the integrals. The solid line is a quadratic curve fitted to the data points. This has the form

$$T = 1.967 - 0.00048x^2 + 0.0000156x^2, \tag{20}$$

where x is the number of primitive basis functions. The shape of the graph suggests that there is a fixed time cost to move the data to and from the FPGA with a value of 1.967 seconds. As larger numbers of integrals are executed, this is amortized to an extent, however, still remains significant for 2000 functions representing 25% of the elapsed time for the whole computation. Table 6 presents the elapsed times in seconds for execution of the integral workload using the C++ std:::future language feature on the CPU. Within the limits of Amdahl's law, the results down each column of the table reflect the scaling enabled by used of additional CPUs. It is interesting to compare the execution of the code as a single for-loop versus using two asynchronous tasks, an experiment that allows us to gauge the cost of using std::async. Table 7 compares these results. It is not surprising that the cost of executing two separate tasks is more expensive for smaller workloads, corresponding to smaller numbers of basis functions. However, by 1000 basis functions, there is an approximate 33% speedup using two tasks.

7 | RELATED WORK

This is the first article, in so far as we are aware to apply heterogeneous accelerator computing to the problem of integral evaluation in electron molecule scattering. GPU computing has previously been successfully applied to computation of molecular bound states where performance gains have of 180× in Hartree-Fock energy + gradient calculations have been observed.[22] The performance of the two electron integral evaluation in this work was studied using three different algorithms, well suited to the GPU architecture and distinct from the many optimal CPU approaches.[15] A combined CPU-GPU algorithm, enabling strong scaling parallelization on inhomogeneous compute clusters, was developed by Kussman and Ochensfeld[23] to evaluate two-electron contributions occurring in Hartree Fock and hybrid density functional theory. Their approach allows an efficient use of CPUs and GPUs simultaneously. They pointed out that the different architectures demand conflicting strategies in order to ensure efficient program execution.

In so far as we are aware, FPGAs have not been used for integration of GTFs in quantum chemistry. On the other hand, there is a large literature on using FPGAs to calculate the quantum mechanical properties of N-body systems, such as in quantum Monte Carlo techniques[24] and in molecular dynamics.[25] The suitability of a hardware approach for molecular dynamics is seen in the Anton system which is a set of customized application specific integrated circuits (ASICs) developed specifically for molecular dynamics simulations on proteins and biological macro-molecules.[26]

8 | CONCLUSIONS AND FUTURE DIRECTIONS

The aim of this article is to examine the viability of applying OpenCL to enable some existing computation kernels on modern heterogeneous accelerators. We chose a FORTRAN program for the evaluation of tail integrals over diffuse Gaussian type functions and converted this into ANSI C++14, using that as a basis from which to develop an OpenCL implementation. The program was taken from a large base of code known as the UK molecular R-matrix suite, a code base that has been in continuous development for 40 years in the UK. This code base was evolved from the IBM Alchemy I program suite for molecular bound state computations, a package on which development began in the Theoretical Chemistry Group at the IBM San Jose laboratory in the late 1960s.

The original code implementation used a kernel function called within a dense set of driver loops. The loop iterations were controlled by the quantum numbers of the diffuse Gaussian functions in the basis set. This analysis is completed based on logical and integer arithmetic and proceeds at very high speed on the CPU. We redesigned the algorithm to a data table driven one, centered on the idea of preparing a table of data where each row required independent execution of a kernel function. The entries in each row of the table are derived from the same driver loops in the original algorithm. The table is copied to global memory on the accelerator before launching the independent evaluation of each row as kernel instances on the accelerator.

We executed this implementation on GPU and FPGA platforms, using an OpenCL SDK specific to each platform, in order to investigate the portability and performance of the implementation. These two accelerators have differing approaches to code execution and we reported on specific optimizations relevant to each architecture. The data-driven approach also works on the host CPU where the kernel instances can be considered as tasks and we have reported two different ways of achieving this. On the one hand, the OpenCL code used on the accelerators can also be used. On the other hand, the C++ language now incorporates mechanisms to easily manage this kind of task based model. The C++ language is adding increasing support for parallelism with the C++17 standard adding support for parallel implementations of the standard template library (STL). Looking forward, a concept known as an executor is proposed as as a basic building block for extension of the C++ language to include heterogeneous accelerators; however, this is not expected to appear until completion of the C++23 standard.[27] However, early implementations of the proposals for executors are beginning to appear, such as in the HPX runtime which is based on C++.[28]

There are clearly a number of ways in which the research we have reported can be extended. The partitioning of operations into tabular data, or even matrix, format with execution of processing functions on each element is a pattern that can be readily identified in many computational science codes. We plan to investigate this approach in other problem domains therefore. A further research angle is to investigate the viability of using dynamic selection of precision, a feature that has been investigated by Martinez and co-workers[29] using GPUs for structure calculations.

ACKNOWLEDGEMENTS

This work has been funded in part by the NanoStreams and AllScale projects. The NanoStreams project is funded by the European Commission under its Seventh Framework Programme as contract number 610509. The AllScale project is funded by the European Commission under its Horizon 2020 Programme as contract number 671603.

ORCID

Charles J. Gillan https://orcid.org/0000-0003-3671-7196

REFERENCES

1. Mihal A, Keutzer K. Mapping concurrent applications onto architectural platforms. In: Jantsch A, Tenhunen H, eds. *Networks on Chips*. 101 Philip Drive Assinippi Park Norwell, MA: Kluwer Academic Publishers; 2003;39-59.
2. Thompson M et al. "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," 2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). Salzburg Austria; 2007; 9-14. https://doi.org/10.1145/1289816.1289823.
3. PGI Visual Fortran user guide. https://www.pgroup.com/resources/docs/18.4/x86/pvf-user-guide/index.htm. December 4th 2019.
4. Morgan LA, Gillan CJ, Tennyson J, Chen X. R-matrix calculations for polyatomic molecules: electron scattering by N_2O. *J Phys B Atomic Mol Opt Phys*. 1997;30:4087796.
5. Burke PG. Collisions with molecules. *R-Matrix Theory of Atomic Collisions: Application to Atomic, Molecular and Optical Processes*. Berlin Heidelberg/Germany: Springer; 2011:533-590.
6. Gillan C, Steinke T, Bock J, Borchert S, Spence I, Scott S. Programming challenges for the implementation of numerical quadrature in atomic physics on FPGA and GPU accelerators. Paper presented at 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing; 2010:757-762; Melbourne, Australia. https://dl.acm.org/doi/10.1109/CCGRID.2010.30. https://www.computer.org/csdl/proceedings-article/ccgrid/2010/4039a757/12OmNBghtqW.
7. Gillan CJ, Steinke T, Bock J, Borchert S, Spence I, Scott NS. Comparing the implementation of two-dimensional numerical quadrature on GPU, FPGA and ClearSpeed systems to study electron scattering by atoms. *Concurr Comput Pract Exper*. 2012;24(1):84-95. doi:10.1002/cpe.1733
8. Gillan CJ, Nagy O, Burke PG, Morgan LA, Noble CJ. Electron scattering by nitrogen molecules. *J Phys B Atomic Mol Phys*. 1987;20:4585.
9. Morgan LA. In: Huo WM, Gianturco FA, eds. *Computational Methods for Electron Molecule Collisions*. New York, NY: Plenum; 1995:227.
10. Tennyson J. Electron molecule collision calculations using the R-matrix method. *Phys Rep*. 2010;491(2?3):29-76.
11. Mašin Z, Benda J, Gorfinkiel JD, Harvey AG, Tennyson J. UKRmol+: a suite for modelling electronic processes in molecules interacting with electrons, positrons and photons using the R-matrix method. *Comput Phys Commun*. 2019;249:107092.
12. Boys SF. Electronic wave functions. I. a general method of calculation for the stationary states of any molecular system. *Proc R Soc Lond Ser A*. 1950;200:542.
13. Davidson Ernest R. Use of double cosets in constructing integrals over symmetry orbitals. *The Journal of Chemical Physics*. 1975;62(2):400. http://dx.doi.org/10.1063/1.430484.
14. Lindh R, Ryu U, Liu B. The reduced multiplication scheme of the Rys quadrature and new recurrence relations for auxiliary function based two-electron integral evaluation. *J Chem Phys*. 1991;95:5889.
15. Ufimtsev IS, Mart?nez TJ. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *J Chem Theory Comput*. 2008;4(2):222-231.
16. Shiozaki T. Call for another Seward: optimization of F12 integral evaluation. *J Unaswered Questions*. 2010;1(1):1-4.
17. Lindh R. Electron repulsion integrals. In: Schaefer HF III, ed. *Encyclopedia of Computational Chemistry*. Vol 4. Chichester, England: Wiley; 1998.
18. McLean AD. In: Lester WA Jr, ed. *Proc. Conf. on Potential Energy Surfaces in Chemistry*. San Jose, CA: IBM; 1971:p87.
19. Almlof J, Taylor PR. In: Dykstra CE, ed. *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules*. Dordrecht, Netherlands: Reidel; 1984.
20. Walker RC, Goetz AW. *Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics*. Hoboken, NJ: Wiley-Blackwell; 2016.
21. Altera SDK for OpenCL; 2016. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. December 5th 2019.
22. Ufimtsev IS, Martinez TJ. Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics. *J Chem Theory Comput*. 2009;5(10):2619-2628.
23. Kussman J, Ochsenfeld C. Hybrid CPU/GPU integral engine for strong-scaling Ab Initio methods. *Chem Theory Comput*. 2017;13(7):3153-3159. https://doi.org/10.1021/acs.jctc.6b01166.
24. Gothandaraman A, Peterson GD, Warren GL, Hinde RJ. FPGA acceleration of a quantum Monte Carlo application. *Parallel Comput*. 2008;34(4?5):278-291.
25. Waidyasooriya HM, Hariyama M, Kashara K. Architecture of an FPGA accelerator for molecular dynamics simulation using OpenCL. Paper presented at: Proceedings of the IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS). Okayama Japan; 2016. https://ieeexplore.ieee.org/document/7550743.
26. Shaw DE, Grossman JP, Bank J, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. (Portland, Oregon). Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2014:41-53; New Orleans, LA, ACM. https://ieeexplore.ieee.org/document/7012191.
27. Hoberock J, Garland M, Kohlhoff C, Mysen C, Edwards C, Brown G. A unified executors proposal for C++, proposal P0443R7, 7th May 2018. http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r7.html.
28. Khatami Z, Troska L, Kaiser H, Ramanujam J, Serio A. HPX smart executors. Paper presented at: Proceedings of the 3rd International Workshop on Extreme Scale Programming Models and Middleware (ESPM2'17); 2017:3; ACM, New York, NY. https://doi.org/10.1145/3152041.3152084
29. Lueher N, Ufimtsev IS, Martinez TJ. Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). *J Chem Theory Comput*. 2011;7(4):949-954. https://doi.org/10.1021/ct100701w.

How to cite this article: Gillan CJ, Spence I. Computing integrals for electron molecule scattering on heterogeneous accelerator systems. *Concurrency Computat Pract Exper*. 2020;e5984. https://doi.org/10.1002/cpe.5984