Title

Achieving performance portability in Gaussian basis set density functional theory on accelerator based architectures in NWChemEx

Authors

Williams-Young, David B

Bagusetty, Abhishek

de Jong, Wibe A

et al.

Publication Date

2021-12-01

DOI

10.1016/j.parco.2021.102829

Peer reviewed

Achieving Performance Portability in Gaussian Basis Set Density Functional Theory on Accelerator Based Architectures in NWChemEx

David B. Williams–Young[a,*], Abhishek Bagusetty[b], Wibe A. de Jong[a], Douglas Doerfler[c], Hubertus J.J. van Dam[d], Alvaro Vazquez-Mayagoitia[e], Theresa L. Windus[f], Chao Yang[a]

[a] *Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA*
[b] *Leadership Computing Facility, Argonne National Laboratory, Argonne, IL 60439, USA*
[c] *National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA*
[d] *Computational Science Initiative, Brookhaven National Laboratory, Upton, NY, 11973, USA*
[e] *Computational Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*
[f] *Department of Chemistry, Iowa State University and Ames Laboratory, Ames, IA 50011, USA*

Abstract

The numerical integration of the exchange-correlation (XC) potential is one of the primary computational bottlenecks in Gaussian basis set Kohn-Sham density functional theory (KS-DFT). To achieve optimal performance and accuracy, care must be taken in this numerical integration to preserve local sparsity as to allow for near linear weak scaling with system size. This leads to an integration scheme with several performance critical kernels which must be hand optimized for each architecture of interest. As the set of available accelerator hardware goes more diverse, a key challenge for developers of KS-DFT software is to maintain performance portability across a wide range of computational architectures. In this work, we examine a modular software design pattern which decouples the implementation details of performance critical kernels from the expression of high-level algorithmic workflows in a device-agnostic language such as C++; thus allowing for developers to target existing and emerging accelerator hardware within a single code base. We consider the efficacy of such a design pattern in the numerical integration of the XC potential by demonstrating its ability to achieve performance portability across a set of accelerator architectures which are representative of those on current and future U.S. Department of Energy Leadership Computing Facilities.

Keywords: Density Functional Theory, Accelerator, Graphics Processing Unit, Performance Portability

1. Introduction

As we approach the inevitable demise of Moore's and Denard's laws, recent years have seen an increasing reliance on the use of accelerators such as graphics processing units (GPUs) to perform the majority of the floating point operations (FLOPs) on contemporary and emerging high-performance computing (HPC) resources [1–3]. As such, there has been a drastic shift in focus for the development of application software to target these new and emerging architectures, especially in the domain of quantum chemisty[4, 5]. However, due to the often complicated workflows and algorithmic designs encompassed by these applications, the effort needed to port high-performance scientific software from CPU- to accelerator-based architectures is often immense. This effort is only compounded as new hardware and programming models are introduced into the HPC ecosystem. As the set of available hardware and associated programming models grows more diverse, an increasing challenge for scientific applications has been to achieve performance portability [6, 7] on existing architectures while minimizing the software development effort needed to target new architectures as they emerge.

Historically, application developers have primarily depended on C, C++ and Fortran for HPC codes. For the multi-core CPU era of computing, this worked well as all HPC class CPUs had

*Corresponding author
Email address: dbwy@lbl.gov (David B. Williams–Young)

compilers for those languages. With the introduction of GPUs into the HPC landscape, the programming model ecosystem has become more diverse. One of the first widely adopted programming languages which targeted GPU architectures was CUDA [8], a language developed by NVIDIA based on C/C++ with extensions to support GPUs. However, CUDA is only supported on NVIDIA's GPUs. While NVIDIA GPUs have been dominant in HPC computing centers, recently AMD and Intel GPUs have both been chosen for three of the first exascale class computing platforms produced for the U.S. Department of Energy (DOE) [9–11]. AMD has chosen HIP as the primary programming model to target its GPUs [12], while Intel has been developing Data Parallel C++ (DPC++) [13]. HIP is very similar (syntactically) to CUDA, while DPC++ is an extension of SYCL [14], which is itself an extension of the ISO standard C++ for targeting heterogeneous architectures. The details of the differences in these programming models is beyond the scope of this paper, but what is important is no one model currently covers the feature set of all three of these architectures completely.

Another option for code developers is to use one the many available declaration based language extensions such as OpenACC [15] and OpenMP [16] or performance portability layers such as Kokkos [17] and RAJA [18]. These programming models are often attractive in that they allow the developer to compose the entireity of their algorithmic workflow in a high-level language and expect to achieve reasonably performant code in the majority of commonly encountered use cases. However, these models offer little recourse in cases when manual optimizations may be exploited to achieve higher performance in critical kernels. Further, the interoperability of these models amongst themselves and with the aformentioned accelerator-specific languages is often a non-trivial endeavour, thus posing considerabile challenges for library developers which aim to target a large number of downstream applications. As such, we do not explictly explore such programming models in this work.

This diverse programming model landscape leaves an application developer with no clear path if their goal is to develop a single code base which can be easily ported from one architecture to another in a performant manner. In this work, we examine a extensible, modular software design pattern which decouples the expression of algorithmic workflows in a high-level language (C++) from the implementation details of its performance critical kernels. As such, each of these performance critical kernels may be expressed in the programming model which is most appropriate for an accelerator architecture of interest. In addition to being a common design pattern typically encountered in object-oriented programming languages such as C++, the concept of algorithm-kernel decoupling has also found great success in the field of numerical linear algebra in performance portable, extensible libraries such as BLIS [19]. Here, we examine how this concept may also be applied to a critical scientific application: Kohn-Sham density functional theory (KS-DFT).

KS-DFT [20] is among the most powerful theoretical techniques for the treatment of quantum phenomena in large, experimentally relevant systems [21, 22]. A primary factor contributing to the success of KS-DFT in computational chemistry and materials science is the existence of highly optimized software implementations which are designed to fully exploit the resources of the compute platforms which they target. Over the years, these efforts have produced highly efficient and massively parallel KS-DFT software for CPU-based (See Ref. [23] and references therein) architectures. More recently, there has been considerable effort afforded to the development of GPU-based KS-DFT software to leverage the latest advances in modern HPC (See Refs. [4, 5, 23] for a review of modern trends as well as Refs. [24–28] for a number of recent developments).

A considerable amount of the computational work incurred by KS-DFT methods is represented by heavy use linear algebra subroutines which are typically provided by optimized BLAS libraries . As such, it is typically justifiable that performance portability may be achieved in large part by ensuring that the application has made use of microarchitecture optimized implementations of critical BLAS operations for the architecture in question, thus delegating the problem of performance portability to the developer of the optimized BLAS library (hardware vendor or otherwise) rather than the developer of the end application. However, as has been recently demonstrated for GPU architectures in the case of Gaussian basis set KS-DFT [24], the efficiency with which batched level-3 BLAS operations may be executed on contemporary GPU architectures shifts the relative computational importance of the BLAS operations on overall time-to-solution to be less than other algorithmic kernels which were previously not dominant. Thus,

on accelerator based architectures, the problem of performance portability in KS-DFT software relies both on the use of optimized BLAS libraries and the hand optimization of specific performance critical kernels for microarchitectures of interest.

In this work, we consider the portable implementation of the Gaussian basis set discretization of KS-DFT in the NWChemEx software package [29, 30]. For atom-centered bases, the formation of the KS-DFT Fock matrix is dominated by the formation of the Coulomb matrix, exchange–correlation (XC) matrix, and the exact-exchange matrix in the case of hybrid KS-DFT methods. Recently, a significant amount of research has been directed to the effecient evalution of the Coulomb and exact exchange matrices in Gaussian basis sets on GPU architectures [27, 31–38], while relatively little effort has been afforded to the evaluation of the XC matrix [24, 25, 27, 28]. As such, we focus on the portable implementation of the XC matrix in this work. The remainder of this work is organized as follows. Sections 2.1 and 2.2 briefly review the necessary algorithmic details required for the numerical assembly of the XC potential on accelerator based architectures. In Sec. 2.3, we examine the specifics of a set of targeted accelerator architectures which are representative of those available on current and proposed for future computational resources provided by DOE leadership computing facilities, and in Sec. 2.4, we outline the proposed modular software framework to achieve performance portability across them. Section 3 outlines some preliminary results regarding the current state of portability achieved using the proposed software framework and Sec. 4 concludes this work with a brief summary and a perspective on future research directions.

2. Methods

2.1. Theory and Background

Within Kohn-Sham density functional theory (KS-DFT) [20, 39], the energetics of quantum many-body interactions are described by the exchange–correlation (XC) energy functional, E^{xc}, which depends on the total electron density, $\rho : \mathbb{R}^3 \to \mathbb{R}$,

$$E^{xc}[\rho] = \int_{\mathbb{R}^3} \varepsilon(\mathbf{r})\rho(\mathbf{r})\mathrm{d}^3\mathbf{r}, \tag{1}$$

where $\varepsilon(\mathbf{r})$ is the XC energy density which depends on the value of ρ at a particular point $\mathbf{r} \in \mathbb{R}^3$. In this work, we consider the generalized gradient approximation (GGA) [40, 41] where ε is described entirely by ρ and its gradient $\nabla\rho : \mathbb{R}^3 \to \mathbb{R}^3$, such that $\varepsilon(\mathbf{r}) \equiv \varepsilon(\rho(\mathbf{r}), \gamma(\mathbf{r}))$ where $\gamma = |\nabla\rho|^2$. However, we note that similar schemes to those presented in this work could be synthesized for other local KS-DFT approximations such as the local density approximation (LDA) and the meta-GGA, as well as non-local approximations such as van der Waals XC functionals [42]. By discretizing ρ / $\nabla\rho$ in a basis set expansion $\mathcal{S} = \{\phi_\mu : \mathbb{R}^3 \to \mathbb{R}\}_{\mu=1}^{N_b}$ given a matrix of expansion coefficients, $\mathbf{P} \in \mathbb{R}^{N_b \times N_b}$ (the *density matrix*), the quantum many-body interactions may be effectively described by a non-linear one-body XC potential, $\mathbf{V}^{xc} \in \mathbb{R}^{N_b \times N_b}$, given by [43–45]

$$V^{xc}_{\mu\nu} = \int_{\mathbb{R}^3} \phi_\mu(\mathbf{r})Z_\nu(\mathbf{r}) + Z_\mu(\mathbf{r})\phi_\nu(\mathbf{r})\mathrm{d}^3\mathbf{r}, \tag{2}$$

where

$$Z_\mu(\mathbf{r}) = \frac{1}{2}\varepsilon_\rho(\mathbf{r})\phi_\mu(\mathbf{r}) + 2\varepsilon_\gamma(\mathbf{r})(\nabla\rho(\mathbf{r})\cdot\nabla\phi_\mu(\mathbf{r})), \tag{3}$$

and $\varepsilon_\rho \equiv \frac{\partial\varepsilon}{\partial\rho}$ and $\varepsilon_\gamma \equiv \frac{\partial\varepsilon}{\partial\gamma}$. In this work, we consider the case when $\mathcal{S}$ consists of atom-centered Gaussian functions ($\phi_\mu(\mathbf{r}) \propto \exp(-\alpha|\mathbf{r}|^2)$) [46], thus $\mathbf{P}$ and $\mathbf{V}^{xc}$ are real-symmetric matrices. The integration of Eqs. (1) and (2) will be referred to as the *XC integration* in the following.

Due to the highly nonlinear character of ε, the XC integrations must be carried out numerically. For atom centered bases, the most commonly adopted approach for the numerical integration is to decompose the full integrand into a sum over atomic integrands such that

$$I[f] \equiv \int_{\mathbb{R}^3} f(\mathbf{r})\,\mathrm{d}^3\mathbf{r} = \sum_{A=1}^{N_A} I_A[f],$$
$$I_A[f] = \int_{\mathbb{R}^3} p_A(\mathbf{r})f(\mathbf{r})\,\mathrm{d}^3\mathbf{r},$$

where N_A is the number of atomic nuclei, and p_A is an atomic partition function which satisfies $\sum_A p_A(\mathbf{r}) = 1\ \forall\mathbf{r} \in \mathbb{R}^3$. Typically, p_A is chosen to form soft boundary Voronoii polyhedra around each of the nuclei [47]. Each atomic integrand is then discretized by a quadrature rule $\mathcal{Q}_A = \{(\mathbf{r}^A_i, w^A_i)\}_{i=1}^{N^A_g}$ with

$$w^A_i = p_A(\mathbf{r}^A_i)\tilde{w}^A_i, \tag{4}$$

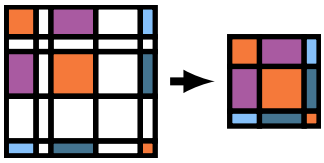

Figure 1: Batch local matrix compression scheme for non-negligible function indices $\mathcal{S}_j$. Colored tiles represent matrix elements which are to be included in the compressed matrix, and white tiles represent matrix elements which are to be neglected.

where $\tilde{w}_i^A$ is the quadrature weight derived directly from the quadrature rule for the approximation of I_A. We denote the total quadrature $\mathcal{Q} = \bigcup_A \mathcal{Q}_A = \{(w_i, \mathbf{r}_i)\}_{i=1}^{N_g}$ with $N_g = \sum_{A=1}^{N_A} N_g^A$ in the following. The algorithmic details regarding the implementation of Eq. (4) are outside the scope of this work, however, we note that this operation scales between $O(N_g)$ and $O(N_A N_g)$ depending on the spacial distribution of the atomic centers and is among the most computationally demanding tasks in the XC integration.

We may exploit the semi-compact nature of the functions in $\mathcal{S}$ by constructing spatially localized quadrature batches

$$\mathcal{Q} = \bigcup_j \mathcal{B}_j, \quad \text{s.t.} \quad \mathcal{B}_j \cap \mathcal{B}_k = \emptyset, \text{ for } j \neq k,$$

such that for every $\mathcal{B}_j$, we may construct a subset of basis functions $\mathcal{S}_j \subset \mathcal{S}$ which are non-negligible within its spacial domain [48]. We denote $N_g^j = |\mathcal{B}_j|$ and $N_b^j = |\mathcal{S}_j|$. For a detailed description of the decomposition of $\mathcal{Q}$ into localized cuboids and subsequent screening of $\mathcal{S}$ employed in this work, we refer the reader to reference [24].

Given a set of quadrature batches $\mathcal{B} = \{\mathcal{B}_j\}$, Eqs. (1) and (2) may be approximated as

$$E^{xc} \approx \sum_{j \in \mathcal{B}} E^{(j)}, \qquad \mathbf{V}^{xc} \approx \sum_{j \in \mathcal{B}} \mathbf{V}^{(j)} \tag{5}$$

where the batch local quantities $E^{(j)}$ and $\mathbf{V}^{(j)}$ may be efficiently assembled via BLAS operations as follows [24],

$$E^{(j)} = \sum_{i \in \mathcal{B}_j} \rho_i^{(j)} \varepsilon_i^{(j)}, \qquad \text{(DOT)} \tag{6}$$

$$V_{\mu\nu}^{(j)} = \sum_{i \in \mathcal{B}_j} \Phi_{\mu i}^{(j)} Z_{\nu i}^{(j)} + Z_{\mu i}^{(j)} \Phi_{\nu i}^{(j)}, \quad \text{(SYR2K)} \tag{7}$$

with $\mu, \nu \in \mathcal{S}_j$. Here $\rho_i^{(j)} \equiv \rho(\mathbf{r}_i)$, $\varepsilon_i^{(j)} \equiv w_i \varepsilon(\mathbf{r}_i)$, and $\Phi_{\mu i}^{(j)} \equiv \phi_\mu(\mathbf{r}_i)$ are the batch local electron density, (quadrature scaled) energy density, and collocation matrix evaluated on the quadrature batch $\mathcal{B}_j$, respectively. The elements of $\mathbf{Z}^{(j)}$ are given by

$$Z_{\mu i}^{(j)} = \frac{1}{2} \varepsilon_{\rho,i}^{(j)} \Phi_{\mu i}^{(j)} + 2 \varepsilon_{\gamma,i}^{(j)} \left(\nabla \rho_i^{(j)} \cdot \nabla \Phi_{\mu i}^{(j)} \right) \tag{8}$$

where $\varepsilon_{\rho/\gamma,i}^{(j)} \equiv w_i \varepsilon_{\rho/\gamma}(\mathbf{r}_i)$. For brevity in the following, we denote the collection of grid local quantities pertaining to a particular $\mathcal{B}_j$ with super-scripted bold-faced symbols, e.g. $\boldsymbol{\rho}^{(j)} = \{\rho_i^{(j)}\}_{i \in \mathcal{B}_j}$.

ρ and $\nabla\rho$ may also be efficiently evaluated using BLAS operations

$$\rho_i^{(j)} = \sum_{\mu \in \mathcal{S}_j} X_{\mu i}^{(j)} \Phi_{\mu i}^{(j)}, \qquad \text{(DOT)} \tag{9}$$

$$\nabla \rho_i^{(j)} = \sum_{\mu \in \mathcal{S}_j} X_{\mu i}^{(j)} \nabla \Phi_{\mu i}^{(j)}, \qquad \text{(DOT)} \tag{10}$$

$$X_{\mu i}^{(j)} = \sum_{\nu \in \mathcal{S}_j} P_{\mu\nu}^{(j)} \Phi_{\nu i}^{(j)}, \qquad \text{(GEMM)} \tag{11}$$

where $\mathbf{P}^{(j)}$ is a packed batch local density matrix which places into contiguous memory the indices contained in $\mathcal{S}_j$ (See Figure 1). We note for clarity that $\mathbf{V}^{(j)}, \mathbf{P}^{(j)} \in \mathbb{R}^{N_b^j \times N_b^j}$ and $\mathbf{Z}^{(j)}, \mathbf{X}^{(j)}, \mathbf{\Phi}^{(j)} \in \mathbb{R}^{N_b^j \times N_g^j}$.

2.2. Algorithm

In this subsection, we briefly outline the algorithmic details of the distributed memory Gaussian basis set XC integration on accelerator based architectures. For a more detailed description of this approach and its implementation on NVIDIA GPU hardware, we refer the reader to reference [24].

The numerical integration scheme described in the previous section decomposes the XC integration into an accumulation of batch local quantities which may be evaluated in parallel. On distributed memory architectures, the quadrature batches may be distributed using a communication-free static load balancing procedure which factors the communication requirements into a single collective reduction (e.g., `MPI_(All)reduce` in the case of MPI message passing) following the digestion of the local work by each independent compute rank [24]. As such, the problem of performance portability in the context of this work amounts to the portable implementation of Eqs. (4) to (11) to digest the quadrature batches assigned to a particular compute rank. To simplify the following discussion, we

Algorithm 1: Evaluation of Quadrature Batches

Input : Quadrature batches $\mathcal{B}$, density matrix $\mathbf{P}$, XC potential $\mathbf{V}^{xc}$, and XC energy E^{xc} all in device memory.
Output: $\mathbf{V}^{xc}$ and E^{xc} updated by quadrature contributions from $\mathcal{B}$

1.1 Update quadrature weights by Eq. (4).
1.2 **batch**$_j$ $\mathbf{P}^{(j)} \leftarrow$ Compress batch local density matrix from $\mathbf{P}$.
1.3 **batch**$_j$ Evaluate $\mathbf{\Phi}^{(j)}/\nabla\mathbf{\Phi}^{(j)}$.
1.4 **batch**$_j$ $\mathbf{X}^{(j)} \leftarrow$ Eq. (11) (VB-GEMM).
1.5 **batch**$_j$ $(\boldsymbol{\rho}^{(j)}, \nabla\boldsymbol{\rho}^{(j)}) \leftarrow$ Eqs. (9) and (10) (F-VB-DOT).
1.6 $(\boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}_\rho, \boldsymbol{\varepsilon}_\gamma) \leftarrow$ Evaluate XC energy density and its derivatives for all points in $\mathcal{B}$.
1.7 Update E^{xc} according to Eqs. (5) and (6) (DOT).
1.8 **batch**$_j$ $\mathbf{Z}^{(j)} \leftarrow$ Eq. (8).
1.9 **batch**$_j$ $\mathbf{V}^{(j)} \leftarrow$ Eq. (7) (VB-SYR2K).
1.10 **batch**$_j$ $\mathbf{V}^{xc} \leftarrow \mathbf{V}^{xc} + \mathbf{V}^{(j)}$.

will assume a 1-to-1 mapping of accelerators to MPI ranks, though 1-to-many and many-to-1 mappings could also be utilized with only minor modifications to the algorithm presented here.

If constructed properly, the amount of work required to evaluate $E^{(j)}$ and $\mathbf{V}^{(j)}$ for a particular $\mathcal{B}_j$ is small. As such, optimiality may be achieved by batching the evaluation of intermediate quantities into individual "batched" kernels as to saturate device resources and mitigate the kernel launch overhead. This procedure is outlined in Alg. 1, where the qualifier **batch**$_j$ indicates a single batched kernel to perform the specified task. We note that Lines 1.1, 1.6 and 1.7 do not have this qualifier as they may be implemented without reference to the batched data structures given that the quadrature batches are internally stored as structures of arrays. Within this batched kernel design, a predetermined number of quadrature batches must be selected to execute concurrently on the device. In this work, we choose to concurrently execute as many batches as will fit in device memory [24]. In addition, this scheme also minimizes the overall impact of inherently serial operations such as data transfers between host and device memory spaces and memory allocations.

In the case of the level-3 BLAS, batching may be achieved by using optimized implementations of batched BLAS operations such as batched SYR2K and batched GEMM [49, 50] for Eqs. (7) and (11) respectively. However, due to the fact that N_b^j and N_g^j can vary significantly between quadrature batches, variable sized batched ("vbatched") level-3 BLAS implementations must be used in this context. We denote these as VB-SY2RK and VB-GEMM in the following. Singular inner products such as Eq. (6) may be implemented using standard accelerated DOT routines. The dot products required for the evaluation of $\boldsymbol{\rho}^{(j)}/\nabla\boldsymbol{\rho}^{(j)}$ could also be implemented using standard DOT routines, but due to the fact that Eqs. (9) and (10) consist of a large number of inner products over a relatively small vector length N_b^j, using such standard implementations in this use case would be inefficient. As with the use of VB level 3 BLAS operations, batching together the DOT operations (VB-DOT) may also improve the throughput for the evaluation of these quantities. However, by recognizing that Eqs. (9) and (10) both involve inner-products of a column of $\mathbf{X}^{(j)}$ with either a columns of $\Phi^{(j)}/\nabla\Phi^{(j)}$, throughput may be further improved by fusing their execution (F-VB-DOT), i.e. loading in a single column of $\mathbf{X}^{(j)}$ and having it persist in memory for its contractions with columns of $\Phi^{(j)}/\nabla\Phi^{(j)}$, thus improving its arithmetic intensity (FLOP/byte) by 25%. In this work, we rely on the use of existing optimized BLAS libraries to carry out the work required for VB-GEMM, VB-SYR2K and DOT. The remaining operations required for the numerical XC integration, including the proposed F-VB-DOT for the batched evaluation of $\boldsymbol{\rho}^{(j)}/\nabla\boldsymbol{\rho}^{(j)}$, must be implemented by user defined kernels for each programming model of interest.

2.3. Considered Accelerators and Programming Models

To expose the most optimization potential in performance critical kernels, we choose to target accelerator hardware with the programming models which specifically target them. In this work, we consider three contemporary accelerators: the NVIDIA Tesla V100 GPU, the AMD Radeon Instinct MI100 GPU, and the Intel Iris Gen9 integrated graphics GPU. We will refer to the Intel GPU simply as the Intel Gen9 in the following. All of these GPUs operate under the SIMT execution model: issuing a single instruction to multiple threads which execute in lock step. The hardware specifics of these GPUs which are relevant to this work may be found in Tab. 1 [51–53].

From a software perspective, there are two primary differences which are of significant consequence: (1) the programming models which most aptly target these GPUs and (2) the linear algebra software stacks which have been optimized for

Table 1: Relevant hardware specifications for the GPU accelerators considered in this work. The device DRAM capacity and peak bandwidth (BW) are given for the slowest memory which is directly accessible to the GPU. DRAM values are not given for Gen9 as they are dependent on the off-chip main memory of the associated compute node

Architecture	DRAM Capacity (GB)	DRAM BW (GB/s)	Peak FP64 (GFLOP/s)	SIMT Length
NVIDIA V100	32	900	7600	32
AMD MI100	64	1200	11500	64
Intel Gen9	–	–	165	32

each architecture. In this work, we target NVIDIA GPUs with the CUDA programming model [8], AMD GPUs with the HIP programming model [12], and Intel GPUs with the DPC++ programming model [13]. For level-1 BLAS operations (DOT), the cuBLAS, hipBLAS, and oneMKL [54] libraries will be used to target NVIDIA, AMD, and Intel GPUs, respectively. Additionally, variable sizes batched level-3 BLAS operations (VB-GEMM and VB-SYR2K) will be provided by the MAGMA [55–57] and hipMAGMA [58] libraries for NVIDIA and AMD GPUs due to their absence in cuBLAS and hipBLAS, respectively. As of this work, oneMKL only provides an API for VB-GEMM, not VB-SYR2K. As Eq. (7) may be alternatively expressed as the sum of two GEMM operations, its evaluation will be performed by two invocations of VB-GEMM for the Intel GPUs in this work, the latter of which applies the transposed GEMM operation of the first.

It is important to note that while both the AMD and NVIDIA GPUs considered in this work are discrete accelerators which exist separate and apart from the CPU which drives it, the Intel GPU is an integrated GPU. As such, both the execution units and the DRAM accessible to the the AMD and NVIDIA GPUs are decoupled from the CPU, whereas only the execution units are decoupled from the CPU on the Intel GPU: the CPU and the GPU share the same DRAM on this accelerator architecture. To hide bandwidth and latency bottlenecks, the Intel GPU internally exhibits a more complicated memory hierarchy than the relatively flat memory models exhibited by discrete GPUs. Management of the memory resources which are shared between the CPU and GPU (DRAM and lowest level cache [LLC]) and transmission of data from the DRAM to the internal GPU memory hierarchy is handled by the Graphics Technology Interface (GTI) on Intel GPUs. The GTI has its own bandwidth for memory transactions into the GPU memory hierarchy separate from the DRAM bandwidth (73.6 GB/s for Gen9).

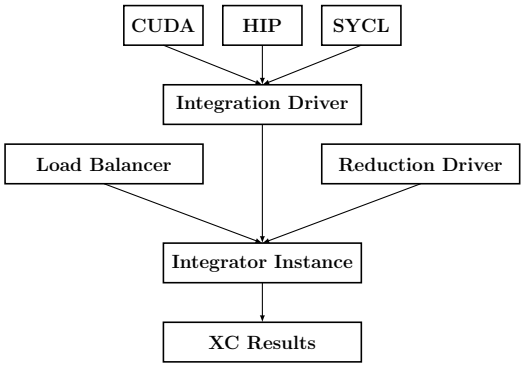

Figure 2: Modular hierarchy of the XC integrator software design proposed in this work. Each node of the hierarchy may be decoupled from the leaves upon which it does not depend and the implementation of dependent nodes is designed to be agnostic from the implementation details of its dependencies.

2.4. Performance Portability by Modular Software Design

In this subsection, we examine a modular software design pattern which allows for high level expression of algorithmic workflows, such as those in Alg. 1, while admitting the potential for opt-in low level optimization of performance critical kernels for specific accelerator hardware with minimal programmer effort. A graphical depiction of the modular hierarchy used in this work is given in Fig. 2. The essence of this design pattern is to decouple the assembly of a particular high-level procedure from the implementation details of its component operations. In so doing, we also obtain an *extensible* framework for which we may provide additional implementations targeting future hardware given the ability to express the individual algorithmic kernels in the programming model which most aptly targets them.

At the highest level, the distributed memory XC integration requires knowledge of three operations: (1) how to effectively balance the computational work on each node to allow for scalability, (2) how to perform the required computation given the work that has been assigned locally, and (3) how to com-

bine these results in some expected way given an underlying message passing implementation (e.g., MPI) and knowledge of where the data must reside on exit (e.g., host or device). From a software development perspective, the implementation details of these three operations are decoupled from one and other. Thus, given an implementation of each of these operations, it is possible to to express the XC integration at a high-level without reference to the implementation details of these operations individually. As such, it is possible to provide specific implementations of each of these tasks for each architecture of interest. However, that does not necessitate that each operation *must* be specialized for each architecture of interest. For example, the distributed reduction phase of the XC integration may be implemented portability using MPI message passing implementations which are aware of device memory spaces. Such an MPI implementation is provided by the Summit supercomputer, and similar implementations have been proposed for future DOE supercomputers equipped with either AMD or Intel GPUs. In such implementations, a single high-level API may be used to perform the reduction whether the results are to reside on the host or device. On the other hand, it is occasionally desirable to implement specific optimizations for these type of collective operations for specific node interconnects, memory consistency models, data localities, etc [59–61]. The modular nature of this software design pattern simply exposes the ability to perform such optimizations when appropriate.

The same logic may also be applied to the computation related to the local work itself: given an implementation of a scheme with which we may manually manage device memory and each of the kernels in Alg. 1, it is possible to construct a procedure to perform the local computation on the device which is decoupled from its implementation on any particular accelerator hardware. However, in the many-programming-model paradigm proposed in the previous subsection, syntactical differences in kernel definition, kernel launch, etc. complicate the implementation of such a decoupled software framework. In practice, such a decoupling may be performed through the exposure of high-level API wrappers which delegate to programming model specific implementation of each of these operations by conditional compilation.

The device memory saturation scheme outlined in reference [24] fundamentally requires knowledge of three operations: (1) obtaining the amount of available memory on a particular device, (2) how to allocate some fraction of that memory, and (3) how to manually perform pool allocations out of preallocated buffers such that reads/writes from this buffer are valid when accessed from the device. For both CUDA and HIP, the first two operations may be fulfilled by the `{cuda,hip}MemGetInfo` and `{cuda,hip}Malloc` routines, the latter of which produces C-style pointers to device memory segments. As of the SYCL 2020 standard (as adopted by DPC++) [14], similar allocation semantics may be achieved using unified shared memory (USM) [62, 63] via the routines `cl::sycl::device::get_info` and `cl::sycl::malloc_device`, respectively. Given the C-pointers to device memory segments from any of the above programming models, pool allocations for the preallocated memory may be implemented in a high-level device-agnostic language (e.g., C/C++) in a straight forward manner. However, care must be taken to ensure that the pool allocations adhere to proper alignment requirements for the accelerator in question.

Modularity in the BLAS components of Alg. 1 may be achieved by conditional delegation to the optimized BLAS libraries which were outlined for each of the considered architectures in the previous subsection. As was previously discussed, each kernel for the non-BLAS components of Alg. 1 must be implemented for each programming model of interest. However, the majority of kernels considered in this work admit generic implementation strategies, i.e. at a high level, the differences in their various implementations for specific programming models are primarily syntactical rather than explicitly architecturally dependent. That is not to say that vendor specific compilers do not produce drastically different low level optimizations for these kernels internally, only that from a kernel specification perspective, the implementations for varying programming models are very similar. For generically implementable kernels, it is often possible to perform source-to-source translations between optimized kernels in a particular programming model to other programming models using external conversion utilities. In this work, we have used the HIPIFY and Intel(R) DPC++ Compatibility Tool (DPCT) [64, 65] to perform the source-to-source translation from CUDA to HIP and DPC++, respectively. For kernels which require further specialization to achieve higher performance on a particular architecture, the aforementioned tools may

also be used as a first pass to generate usable kernels in a programming model of interest and then optimizations may be applied on a per-kernel basis.

3. Results and Discussion

In this section, we examine the extent to which the proposed modular software design pattern is able to achieve performance portability across the considered accelerator architectures. Calculations involving the NVIDIA V100 GPU were performed on the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF). CUDA programs were produced using the `nvcc` compiler and cuBLAS from the CUDA 11 SDK and VB-GEMM/SYR2K were provided by MAGMA 2.5.3. Calculations involving the AMD Radeon Instinct MI100 GPU were performed on the Tulip test bed provided by Hewlett Packard Enterprise (HPE). HIP programs were produced using the `hipcc` compiler and hipBLAS from the Radeon Open Compute Platform (ROCm) 3.8.0 SDK and VB-GEMM/SYR2K were provided by hipMAGMA 2.0.0 [58]. Calculations involving the Intel Iris Gen9 integrated GPU were performed on the Joint Laboratory of System Evaluation (JLSE) testbed at the Argonne Leadership Facility (ALCF). Each JLSE Iris node consists of a Intel Xeon E3-1585 v5 CPU with an integrated Gen 9 GPU and 64 GB DDR4 @ 2133MHz with two memory channels [53], leading to a peak bandwidth of 34 GB/s accessible to the GPU. DPC++ programs were produced using the `dpcpp` compiler from the oneAPI SDK (Intel(R) oneAPI Pro 2021.1 (pre-release) beta10). DPC++ programs may be executed by multiple backends using the Intel(R) Graphics Compute Runtime. In this work, we utilize the Intel(R) Level Zero runtime (driver version 0.3) to execute DPC++ programs.

3.1. Performance Critical Kernels

In this work, we examine the portable implementation of three representative kernels:

1. the evaluation of the partition weights (Eq. (4) and Line 1.1),
2. the evaluation of the batch-local collocation matrix and its gradient ($\mathbf{\Phi}^{(j)}/\nabla\mathbf{\Phi}^{(j)}$, Line 1.3), and
3. the evaluation of the batch-local $\boldsymbol{\rho}^{(j)}/\nabla\boldsymbol{\rho}^{(j)}$ via F-VB-DOT (Eqs. (9) and (10) and Line 1.5).

Each of these kernels are FLOP intensive kernels which have been previously demonstrated to be performance critical for the XC integration on accelerator architectures [24]. The details of the programming-model specific optimizations applied to these kernels is beyond the scope of this work, however the general schema of these kernels which are common to all of the considered models may be described as follows: (1) is a 1-dimensional (1D) kernel where each quadrature point is assigned to a particular thread, and all quadrature points may be scheduled independently on the device. SIMT divergences may be kept to a minimum by ensuring that quadrature points which are spatially close are stored contiguously in device memory. (2) is a 2D kernel: each pair of basis functions and quadrature points is assigned to a particular thread on a 2D process grid. To minimize SIMT divergences, the evaluation of quadrature points for the same basis function are restricted to the same SIMT unit. (3) is also a 2D kernel, where each quadrature point is assigned to a particular SIMT unit and the DOT component of the F-VB-DOT reduction in Eqs. (9) and (10) is applied along the SIMT unit.

Due to the difference in computational power exhibited by the considered GPUs accelerators, absolute measures of performance on any particular architecture does not necessarily yield any information about how the achieved performance is portable between architectures. Instead, relative measures of performance are often more instructive [6, 7]. Figure 3 illustrates the relative importance of the aforementioned kernels together with the performance critical BLAS operations (VB-GEMM and VB-SYR2K) for a representative test problem: the numerical integration of the XC potential for the Taxol molecule at the RPBE0/6-31G(d) level of theory. We have used the same molecular geometry as the one used for the same problem in reference [24]. A hard cap of $N_g^j \leq 512$ was imposed in the construction of the quadrature batches which yielded batch sizes ranging from $30 \leq N_g^j \leq 512$ and $5 \leq N_b^j \leq 819$. To improve memory access patterns and in the evaluation of the collocation and density kernels, row-major data structures were employed for the batch-local matrices on each of architectures considered. Qualitatively, we can see that each of these kernels consumes roughly the same percentage of the overall calculation time across each of the considered architectures, indicating that the computational bottlenecks are more or less the

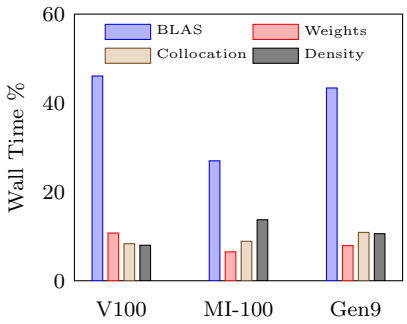

Figure 3: Percentage of overall XC integration wall time spent on performance critical kernels for the considered accelerators for Taxol RPBE0/6-31G(d). The BLAS percentage includes all invocations of GEMM and SYR2K, DOT is negligible for all accelerators considered.

same on each of these GPUs.

3.2. Roofline Performance Characterization

In order to understand the performance of an application on a particular architecture, it is often desirable to examine the performance relative to the peak performance of the architecture in question (i.e. its architectural efficiency) rather than examining absolute performance metrics such as time-to-solution [6, 7]. In addition, this allows for the meaningful comparison of the performance of an application across several architectures, especially when those architectures admit drastically different computational power. However, the peak computational performance of a particular architecture cannot be characterized by a single metric due to the existence of many competing bottlenecks on modern architectures: primarily computational bottlenecks which depend on the maximum rate a processor can issue instructions and perform arithmetic operations, and memory bandwidth bottlenecks which depend on the speed with which data can be accessed from the memory hierarchy and made available to the processor. In this work, we examine the architectural efficiency of the performance critical kernels outlined in the previous subsection through the use of the Roofline Performance Model (RPM) [66, 67].

The RPM characterizes the peak performance of a system to be a function of an operational intensity: the ratio of the number of operations performed to the memory traffic required to perform said operations. If the operations of interest are related to floating point arithmetic (ADD/MUL, FMA, etc), this intensity is referred to as an arithmetic intensity (AI) and is measured in FLOP/Byte [67]. Alternatively, the RPM may be characterized in terms of instructions issued by the processor, in which case this intensity is referred to as an instructional intensity (II) and is measured in instructions/Byte [68]. Given an operational intensity (OpI) of interest, the peak achievable throughput (OpR) for that operation may be expressed as

$$\mathrm{OpR}(\mathrm{OpI}) = \min \begin{cases} \text{Peak OpR} \\ \text{BW} \times \mathrm{OpI} \end{cases} \tag{12}$$

where "peak OpR" and "BW" are the peak operational throughput and bandwidth of the architecture in question. In the Intel Gen9 architecture where the GPU and LLC are separated by the GTI interconnect, peak achievable throughputs may be modeled through adding an additional memory roof to the RPM consisting of the peak bandwidth exhibited by the Gen9 memory hierarchy. In particular, if the data working set for a particular kernel completely resides in LLC, or there is extensive reuse, its performance is bounded by the BW associated with the GTI rather than the DRAM. In this work, the Empirical Roofline Toolkit (ERT) was used to determine the roofline parameters for each of the GPUs used in this study [69], the results of which are shown in Table 2.

Table 2: Empirical roofline peaks collected by the ERT for each of the considered accelerators.

Architecture	Roofline Peaks	
	DRAM BW (GB/s)	Peak OpR
NVIDIA V100	789.2	7796.6 GFLOP/s
AMD MI100	891.3	4792.6 GInst/s
Intel Gen9	27.6	157.9 GFLOP/s

NVIDIA. For the NVIDIA V100, we used the NVIDIA's `nvprof` tool to obtain kernel-level performance metrics. Double-precision (FP64) floating-point operations were counted with `flop_count_dp`. Read and write transactions were measured using `dram_read_transactions` and `dram_write_transactions` respectively. Each transaction contains 32 bytes. Kernel performance characterization for the V100 have been made using arithmetic intensity as the operational intensity, yielding the FP64 DRAM roofline characterization of the aforementioned kernels in Fig. 4.

AMD. For the AMD MI100, the `rocprof` profiling tool was used to obtain kernel-level performance metrics. As `rocprof` does not provide support to count floating-point operations, instructional intensity has been used as the operative operational

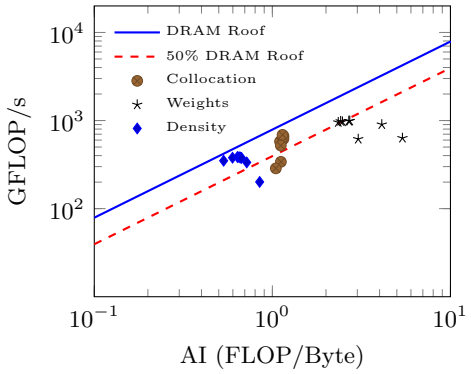

Figure 4: FP64 kernel roofline for considered kernels on the NVIDIA Tesla V100. Each data point represents a single batched kernel invocation.

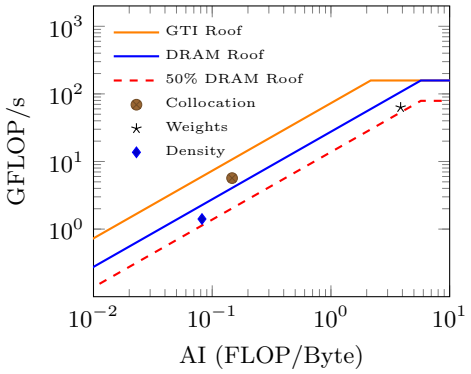

Figure 6: FP64 kernel roofline for considered kernels on the Intel Gen9. Each data point represents the average performance of the specified kernel over all invocations as is returned by Intel(R) Advisor.

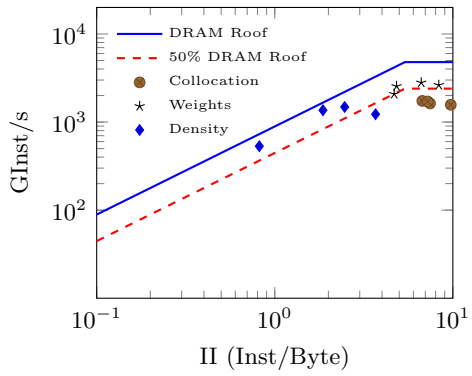

Figure 5: FP64 kernel roofline for considered kernels on the AMD Radeon Instinct MI100. Each data point represents a single batched kernel invocation.

intensity for the performance characterizations on these GPUs. As such, we may not use the standard results from ERT to construct our empirical roofline model. The details of how to construct such a roofline are outside the scope of this work, however we have made the software required to perform such an empirical measurement publicly available [70]. To measure the operation count for each kernel, we have used `VALUInsts` counter, which is the average number of vector ALU instructions executed per work item. Read and write data counts to global memory were measured with `FetchSize` and `WriteSize` respectively. These counts are expressed in kibibytes (KiB). The instruction based roofline performance characterization for the aforementioned kernels is given in Fig. 5 for the AMD MI100.

Intel. For the Intel Gen9 GPU, we have used the Intel(R) Advisor 2021.1 beta10 (build 607346) Command Line Tool to collect kernel-level performance metrics. Unlike `nvprof` and `rocprof` which collect metrics separately on a per-invocation basis, the Intel Advisor automates the collection of metrics related to the AI and FLOP counts into a single invocation with the arguments `--collect=roofline --profile-gpu`. However, rather than reporting results for each kernel invocation, it reports an average performance metric for each kernel. The FP64 roofline performance characterization for the aforementioned kernels in terms of both the DRAM and GTI RPM roofs is given in Fig. 6 for the Intel Gen9 GPU. Remark that the collocation kernel exceeds the DRAM roof and is bounded by the GTI roof. We discuss this phenomena in the following.

A summary of the average architectural efficiencies (AE, the ratio of achieved to peak operational throughput) using these accelerators specific strategies is given in Tab. 3. As the Intel and NVIDIA characterizations were both made using AI as the

Table 3: Average architectural efficiencies of performance critical kernels for Taxol RPBE0/6-31G(d). Architectural efficiencies were computed per-kernel invocation relative to Eq. (12) for the operational intensity measured for each architecture and then averaged over the kernel invocations.

Architecture	Average Architectural Efficiency (%)		
	Collocation	Weights	Density
NVIDIA V100	60.98	39.18	64.45
AMD MI100	34.67	55.58	64.90
Intel Gen9	52.48	59.10	46.40

operative operational intensity, their results are directly comparable. For the weights kernel, which is a 1D kernel that admits simple memory access patterns, both CUDA and DPC++ produce kernels with similar AI and DPC++ achieves higher AE. For the 2D kernels (Density and Collocation) which exhibit more complicated memory access patterns, DPC++ produces kernels with roughly 0.1x the AI of those produced by CUDA. Due to the fact that Gen9 and the V100 admit the same SIMT length, the number of FLOPs per SIMT unit would be expected to be roughly the same between the two architectures. Thus the difference in AI between the two architectures is likely due to the fact that the CUDA compiler produces more aggressive optimizations for the memory access patterns in these kernels than are produced by the DPC++ compiler and the Level Zero runtime. A notable feature of the DPC++ performance characterization is the fact that the collocation kernel is not bounded by the DRAM roof, but rather the GTI roof. This indicates that the collocation kernel is able to reuse data populated in the LLC shared between the CPU and GPU in the Gen9 architecture. This is a reasonable observation due to the fact that the set of basis function parameters $\{\mathbf{R}_\mu, L_\mu, \mathcal{G}_\mu\}$ may be reused for each quadrature point for which ϕ_μ is evaluated, thus being cached into the LLC when read for the first evaluation. The observed AE for this kernel would likely be improved by tuning the kernel invocation to ensure these parameters can reside in LLC for a particular subset of $\mathcal{S}$ without being ejected over the course of the kernel execution.

On the other hand, performance characterizations for the AMD MI100 were made using II rather than AI. As such, it is not meaningful to compare these AE's directly, i.e. X% efficiency with respect to AI does not necessarily indicate X% efficiency with respect to II. That being said, these two methods provide compatible information as they both produce valid performance characterizations of the kernels of interest, relative to their architectural bounds. Both the Weights and Density kernels achieved between 50%-70% architectural efficiency on average for the MI100 which is comparable to the efficiencies achieved on the Intel and NVIDIA GPUs. The collocation kernel on the other hand under performed on the MI100 using this performance characterization. The reasons for this underperformance in terms of the II characterization is not immediately clear without access to additional performance metrics on AMD hardware. However, due to the fact that the collocation kernel consumes roughly the same percentage of overall compute time across all of the considered architectures, it is likely that this low performance characterization is an artifact of the II performance model rather than something inherent in the kernel implementation itself.

4. Conclusion

As GPU hardware grows more diverse, scientific application developers must develop portable, extensible software in order to fully exploit the computational power of modern HPC resources. However, with this growing diversity in hardware comes an additional challenge in that no one programming model exposes the entire feature set of existing or future accelerator architectures. Thus, to achieve optimal performance on an architecture of interest, it is typically the case that one must target specific hardware with the programming model which, from either a language or compiler perspective, most aptly exposes the most optimization potential. In this report, we examined the problem of performance portability for the implementation of KS-DFT in the NWChemEx software package.

We proposed a modular software framework which decouples the expression of the algorithmic workflow for the numerical integration of the XC potential from the implementation details of its performance critical kernels. As such, the overall algorithmic scaffold of this task may be expressed in a high-level language (C++ in this work) while allowing for the expression of each of the kernels in an accelerator specific programming model of choice. In practice, this allows the developer to maintain a single code base which targets multiple accelerators simultaneously with the ability to perform per-architecture optimizations of performance critical operations without affecting the implementations on other architectures and may be extended to future architectures simply by adding implementations of the appropriate kernels in the programming model which is most aptly applicable. This modularity has been made efficacious in large part to the existence of architecturally optimized BLAS libraries such as MAGMA, hipMAGMA and oneMKL.

We have demonstrated that the proposed software framework allows for sufficient flexibility in

kernel specialization to exhibit reasonable performance portability across a diverse set of accelerator architectures: the NVIDIA V100 GPU, the AMD MI100 GPU, and the Intel Iris Gen9 integrated graphics GPU. Due to the large difference in computational power exhibited by these architectures, we have chosen to characterize the performance of the XC integration in terms of relative performance metrics rather than absolute metrics such as time-to-solution. Qualitatively, we have demonstrated that the proposed scheme exhibits the same bottlenecks across architectures, thus indicating that the implementation strategies used for the performance critical kernels examined in this work are generally portable across the considered programming models. Quantitatively through the use of the Roofline Performance Model, we have demonstrated that the performance critical kernels that must be optimized for each architecture of interest achieve reasonably portable architectural efficiency using the proposed scheme.

Although promising, the results presented in this work are largely preliminary leaving many areas for improvement and future research. In particular, the source-to-source translation strategy used in this work to convert optimized kernels in CUDA to either HIP or DPC++ will typically not yield the most optimal implementation strategies across different programming models. However, the decoupling of algorithmic specification from kernel implementation, admitted by design pattern proposed in this work, allows for fine-grained tuning of each kernel individually for each programming model of interest. As such, a major direction of future research posed by this work would be to perform architecture specific optimizations for performance critical kernels. In addition, there were two kernels of importance which were not explicitly examined in this work. The first is the kernel used to pack/unpack the batch-local matrices as depicted in Fig. 1. This kernel can become dominant for large problems on all of the architectures considered (see Ref. [24] for examples on NVIDIA hardware). This kernel was omitted due to the fact that the cross-over point in system size for which it becomes dominant over the FLOP intensive kernels examined in this work is far past the size which is currently practical on the Gen9 GPU, and thus would not provide any additional insight into the problem of performance portability in this context. The second is the evaluation of the XC energy density and its derivatives on the device. This kernel was omitted primarily due to the fact that it is not performance critical ($<0.1\%$ of overall runtime on all of the considered accelerators). However, due to the vast number of approximate XC functionals which exist in the literature, the development of a device portable library to perform this task poses a non-trivial software development effort. The development of such a library will be addressed in a future publication by the authors.

Acknowledgement

The authors would like to acknowledge Thomas Applencourt and Michael D'mello from the Argonne Leadership Computing Facility for many helpful discussions regarding SYCL, Intel DPC++ and the Intel oneAPI software stack related to this work. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This Work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory and AB acknowledges this work as associated with ALCF Aurora Early Science Program.

References

[1] Erich Strohmaier, Highlights of the 55th TOP500 List, Slides 30,31; Accessed: 2020-10-23.
URL `https://top500.org/media/filer_public/54/77/5477d858-1f1e-410b-994b-b7122cfd1d57/top500_2020_06_v2_web.pdf`

[2] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, W.-m. Hwu, Gpu clusters for high-performance computing, in: 2009 IEEE International Conference on Cluster Computing and Workshops, IEEE, 2009, pp. 1–8.

[3] L. A. Parnell, D. W. Demetriou, V. Kamath, E. Y. Zhang, Trends in high performance computing: Exascale systems and facilities beyond the first wave, in: 2019 18th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2019, pp. 167–176.

[4] M. S. Gordon, G. Barca, S. S. Leang, D. Poole, A. P. Rendell, J. L. Galvez Vallejo, B. Westheimer,

Novel computer architectures and quantum chemistry, J. Phys. Chem. A 124 (23) (2020) 4557–4582.
[5] M. S. Gordon, T. L. Windus, Editorial: Modern architectures and their impact on electronic structure theory, Chem. Rev. 120 (17) (2020) 9015–9020.
[6] S. J. Pennycook, J. D. Sewall, V. W. Lee, A metric for performance portability (2016). arXiv:1611.07409.
[7] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, J. Salmon, Performance portability across diverse computer architectures, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).
[8] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st Edition, San Francisco, CA, USA, 2012.
[9] Frontier, ORNL's Exascale Supercompter, accessed: 2020-10-23.
URL https://www.olcf.ornl.gov/frontier/
[10] Aurora, ALCF's Exascale Supercomputer, accessed: 2020-10-23.
URL https://www.alcf.anl.gov/aurora
[11] El Capitan, LLNL Exascale Supercomputer, accessed: 2020-10-23.
URL https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer
[12] HIP Documentation, accessed: 2020-10-23.
URL https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html
[13] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, J. Sewall, Data parallel c++: Enhancing sycl through extensions for productivity and performance, in: Proceedings of the International Workshop on OpenCL, IWOCL '20, 2020.
[14] SYCL™ SpecificationGeneric heterogeneous computing for modern C++, Version 2020 provisional, accessed: 2020-10-23.
URL https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf
[15] OpenACC.org: More Science, Less Computing, accessed: 2020-10-23.
URL https://www.openacc.org
[16] OpenMP.org: The OpenMP API specification for parallel programming, accessed: 2020-10-23.
URL https://www.openmp.org
[17] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel. Distrib. Comput. 74 (12) (2014) 3202 – 3216.
[18] D. A. Beckingsale, J. Burmark, R. Hornung, et al, Raja: Portable performance for large-scale scientific applications, in: 2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc), IEEE, 2019, pp. 71–81.
[19] F. G. Van Zee, R. A. van de Geijn, BLIS: A framework for rapidly instantiating BLAS functionality, ACM TOMS 41 (3) (2015) 14:1–14:33.
[20] W. Kohn, L. J. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. 140 (1965) A1133–A1138.
[21] L. E. Ratcliff, S. Mohr, G. Huhs, T. Deutsch, M. Masella, L. Genovese, Challenges in large scale quantum mechanical calculations, WIREs Comput. Mol. Sci. 7 (1) (2017) e1290.
[22] X. Wu, F. Kang, W. Duan, J. Li, Density functional theory calculations: A powerful tool to simulate and design high-performance energy storage and conversion materials, Progress in Natural Science: Materials International 29 (3) (2019) 247–255.
[23] C. D. Sherrill, D. E. Manolopoulos, T. J. Martínez, A. Michaelides, Electronic structure software, J. Chem. Phys. 153 (7) (2020) 070401.
[24] D. B. Williams-Young, W. A. de Jong, H. J. van Dam, C. Yang, On the efficient evaluation of the exchange correlation potential on graphics processing unit clusters, Frontiers in Chemistry 8 (2020) 951.
[25] M. Manathunga, Y. Miao, D. Mu, A. W. Götz, K. M. Merz, Parallel implementation of density functional theory methods in the quantum interaction computational kernel program, J. Chem. Theory Comput. 16 (7) (2020) 4315–4326.
[26] W. P. Huhn, B. Lange, V. W. zhe Yu, M. Yoon, V. Blum, Gpu acceleration of all-electron electronic structure theory using localized numeric atom-centered basis functions, Comput. Phys. Commun. 254 (2020) 107314.
[27] J. Kussmann, H. Laqua, C. Ochsenfeld, Highly efficient resolution-of-identity density functional theory calculations on central and graphics processing units, J. Chem. Theory Comput. 17 (3) (2021) 1512–1521. doi:10.1021/acs.jctc.0c01252.
[28] M. Manathunga, C. Jin, V. W. D. Cruzeiro, et al, Harnessing the power of multi-gpu acceleration into the quantum interaction computational kernel program, J. Chem. Theory Comput. 17 (7) (2021) 3955–3966.
[29] K. Kowalski, E. Apra, R. Bair, et al, From nwchem to nwchemex: Evolving with the computational chemistry landscape, Chem. Rev. 121 (8) (2021) 4962–4998.
[30] E. Aprà, E. J. Bylaska, W. A. de Jong, N. Govind, et al., Nwchem: Past, present, and future, J. Chem. Phys. 152 (18) (2020) 184102.
[31] G. M. J. Barca, J. L. Galvez-Vallejo, D. L. Poole, A. P. Rendell, M. S. Gordon, High-performance, graphics processing unit-accelerated fock build algorithm, J. Chem. Theory Comput. 16 (12) (2020) 7232–7238.
[32] H. Laqua, T. H. Thompson, J. Kussmann, C. Ochsenfeld, Highly efficient, linear-scaling seminumerical exact-exchange method for graphic processing units, J. Chem. Theory Comput. 16 (3) (2020) 1456–1468.
[33] J. Kalinowski, F. Wennmohs, F. Neese, Arbitrary angular momentum electron repulsion integrals with graphical processing units: Application to the resolution of identity hartree–fock method, J. Chem. Theory Comput. 13 (7) (2017) 3160–3170.
[34] N. Luehr, A. Sisto, T. J. Martínez, Gaussian Basis Set Hartree–Fock, Density Functional Theory, and Beyond on GPUs, 2016, Ch. 4, pp. 67–100.
[35] I. S. Ufimtsev, T. J. Martínez, Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation, J. Chem. Theory Comput. 4 (2) (2008) 222–231.
[36] I. S. Ufimtsev, T. J. Martinez, Quantum chemistry on graphical processing units. 2. direct self-consistent-field implementation, J. Chem. Theory Comput. 5 (4) (2009) 1004–1015.
[37] Y. Miao, K. M. Merz, Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations, J. Chem. Theory Comput. 9 (2) (2013) 965–976.
[38] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, T. L. Windus, Uncontracted rys quadrature

implementation of up to g functions on graphical processing units, J. Chem. Theory Comput. 6 (3) (2010) 696–704.
[39] R. G. Parr, W. Yang, Density Functional Theory of Atoms and Molecules, Vol. 16 of International Series of Monographs on Chemistry, 1994.
[40] J. P. Perdew, W. Yue, Accurate and simple density functional for the electronic exchange energy: Generalized gradient approximation, Phys. Rev. B 33 (1986) 8800–8802.
[41] J. P. Perdew, Density-functional approximation for the correlation energy of the inhomogeneous electron gas, Phys. Rev. B 33 (1986) 8822–8824.
[42] J. Hermann, A. Tkatchenko, Electronic exchange and correlation in van der waals systems: Balancing semilocal and nonlocal energy contributions, J. Chem. Theory Comput. 14 (3) (2018) 1361–1369.
[43] A. Petrone, D. B. Williams-Young, S. Sun, T. F. Stetina, X. Li, An efficient implementation of two-component relativistic density functional theory with torque-free auxiliary variables, Eur. Phys. J. B 91 (7) (2018) 169.
[44] A. M. Burow, M. Sierka, Linear scaling hierarchical integration scheme for the exchange-correlation term in molecular and periodic systems, J. Chem. Theory Comput. 7 (10) (2011) 3097–3104.
[45] K. Yasuda, Accelerating density functional calculations with graphics processing unit, J. Chem. Theory Comput. 4 (8) (2008) 1230–1236.
[46] J. A. Pople, P. M. Gill, B. G. Johnson, Kohn—sham density-functional theory within a finite basis set, Chem. Phys. Lett. 199 (6) (1992) 557 – 560.
[47] A. D. Becke, A multicenter numerical integration scheme for polyatomic molecules, J. Chem. Phys. 88 (4) (1988) 2547–2553.
[48] R. E. Stratmann, G. E. Scuseria, M. J. Frisch, Achieving linear scaling in exchange-correlation density functional quadratures, Chem. Phys. Lett. 257 (3-4) (1996) 213–223.
[49] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Batched matrix computations on hardware accelerators based on gpus, IJHPCA 29 (2) (2015) 193–208.
[50] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Performance, design, and autotuning of batched gemm for gpus, in: High Performance Computing, 2016, pp. 21–38.
[51] NVIDIA V100 Architecture Specification, accessed: 2021-07-13.
URL https://www.nvidia.com/en-gb/data-center/tesla-v100/
[52] AMD MI100 Architecture Specification, accessed: 2021-07-13.
URL https://www.amd.com/en/products/server-accelerators/instinct-mi100
[53] Intel Xeon E3-1585 v5 Architecture Specification, accessed: 2021-07-13.
URL https://ark.intel.com/content/www/us/en/ark/products/93742/intel-xeon-processor-e3-1585-v5-8m-cache-3-50-ghz.html
[54] oneAPI Math Kernel Library Specification, accessed: 2020-10-23.
URL https://spec.oneapi.com/versions/latest/elements/oneMKL/source/index.html
[55] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Computing 36 (5-6) (2010) 232–240.
[56] R. Nath, S. Tomov, J. Dongarra, An Improved MAGMA GEMM For Fermi Graphics Processing Units, Int. J. High Perform. Comput. Appl. 24 (4) (2010) 511–515.
[57] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, S. Tomov, High-Performance Tensor Contractions for GPUs, Tech. Rep. UT-EECS-16-738 (01-2016 2016).
[58] C. Brown, A. Abdelfattah, S. Tomov, J. Dongarra, hipmagma v2.0.0 (Jul. 2020).
[59] H. Shan, S. Williams, C. W. Johnson, Improving mpi reduction performance for manycore architectures with openmp and data compression, in: 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2018, pp. 1–11.
[60] K. Ibrahim, Optimizing breadth-first search at scale using hardware-accelerated space consistency, in: 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC), 2019, pp. 23–33.
[61] K. Ibrahim, Cspacer: A reduced api set runtime for the space consistency model, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2021, p. To Appear.
[62] M. Mrozek, B. Ashbaugh, J. Brodman, Taking memory management to the next level: Unified shared memory in action, in: Proceedings of the International Workshop on OpenCL, 2020, pp. 1–3.
[63] D.-A. Constantinescu, A. Navarro, F. Corbera, J.-A. Fernández-Madrigal, R. Asenjo, Efficiency and productivity for decision making on low-power heterogeneous cpu+ gpu socs, J. Supercomput. (2020) 1–22.
[64] Intel DPC++ Compatibility Tool Developer Guide and Reference , accessed: 2020-10-23.
URL https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top.html
[65] S. Christgau, T. Steinke, Porting a legacy cuda stencil code to oneapi, in: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2020, pp. 359–367.
[66] S. Williams, Auto-tuning performance on multicore computers, Ph.D. thesis, EECS Department, University of California, Berkeley (December 2008).
[67] S. Williams, A. Watterman, D. Patterson, Roofline: An insightful visual performance model for floating-point programs and multicore architectures, Communications of the ACM (April 2009).
[68] N. Ding, S. Williams, An instruction roofline model for gpus, in: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), IEEE, 2019, pp. 7–18.
[69] Roofline Toolkit, Accessed: 2020-10-27.
URL https://crd.lbl.gov/departments/computer-science/par/research/roofline/software/ert/
[70] An Instruction Based Roofline Method for AMD GPUs, Accessed: 2020-10-31.
URL https://bitbucket.org/dwdoerf/amd-gpu-roofline-method