Strategies to Accelerate VASP with GPUs Using OpenACC

Stefan Maintz[1] and Markus Wetzstein[2]
1) NVIDIA, Munich, Germany. E-mail: smaintz@nvidia.com
2) NVIDIA, Zurich, Switzerland. E-mail: mwetzstein@nvidia.com

***Abstract*—We present results of a porting effort of VASP (the Vienna Ab Initio Simulation Package) to GPUs, using OpenACC. While having been useful to researchers, the existing CUDA C based port of VASP was hard to maintain due to source code duplication. We demonstrate a directive based OpenACC adaptation for the most important DFT-level solvers available in VASP: RMM-DIIS and blocked-Davidson. A comparative performance study shows that the OpenACC efforts can even significantly outperform the former port. No extensive code refactoring was necessary. Guidelines to managing device memory for heavily aggregated data structures are presented. These lead to cleaner code and lower the entry barrier to accelerate additional parts of VASP and might prove useful for accelerating other high-performance applications.**

Keywords-GPU; VASP; OpenACC; CUDA; porting; performance optimization; heterogeneous; maintainability

I. INTRODUCTION

With the advent of GPUs for general purpose computing, endeavors started to accelerate quantum-chemistry codes, which consume a dominant share on computing centers all over the world. Among these, the Vienna ab-initio simulation package (VASP) [1-6] is one of the most widely employed programs. When it comes to plane-wave based solid-state calculations, it is perhaps the most important one. In 2015 at NERSC, VASP alone consumed about 12% of the total computing cycles [7] and the situation is similar at other HPC centers. Hence, it is not surprising that starting in 2011, first success stories to accelerate parts of VASP have been published [8-10]. Each of these efforts focused on different algorithms, solvers and even levels of theory. Given the plethora of such combinations that are available in VASP, even the combined and extended port that officially became part of the VASP release 5.4.1 later in 2015, could not cover all of these. This is understandable given all the available pathways through the code, many of which show profiles with lots of kernels with comparable importance for the total run time. Thus, optimization and porting to heterogenous architectures is a challenge. For such cases, Amdahl's law dictates to focus on whole application performance and requires porting large portions of the code. However, given a limited amount of development time this will eventually lead to a trade-off between fully optimizing specific kernels versus offloading others to the accelerator at all.

Starting with release 5.4.4, VASP officially supports being compiled with the PGI compiler suite version 16.10 and higher. This is one option to enable the use of programming frameworks like CUDA Fortran or OpenACC, which allow to target NVIDIA GPUs and integrate seamlessly with Fortran, the programming language used for VASP. With respect to the described timeframes above, the developers of the first GPU-accelerated versions of VASP did not have these options. They chose to base their efforts on CUDA C kernels and utilizing wrapper functions that can be linked and called from Fortran. Interception points were introduced in the call tree to hand off some computation to the GPU and then proceed with the result.

While this proved valuable to show the performance potential of GPUs, it led to massive code duplication and significantly increased maintenance cost for the GPU port. The VASP CPU code is continuously updated and enhanced to enable more and improved science, and these changes needed to be manually integrated into the GPU call-tree as well. With limited staffing resourced for development, keeping the existing code base up to date and porting new features to the GPU becomes difficult, in particular in the context of achieving good overall performance in the first place.

From our perspective, a directive-based approach like OpenACC combined with interfacing to specialized and vendor-optimized libraries seemed like an ideal solution here. This approach utilizes the knowhow of compiler and library engineers for generating well optimized low-level code. It allows the application developer to concentrate on the scientific problems to be solved, and to drop into low-level programming only on an as-needed basis. Our goal in this effort was to make such a vision a reality, and to understand what it takes to apply it to software packages as complex as VASP.

Following this strategy, our ultimate goal was to substantially improve the maintainability of the code base while making it feasible to port new parts of the code in the future. To achieve this in our feasibility study, our port was based on a development snapshot of VASP. This minimized the risk of diverging the code base right from the start. The port had to work out of the same existing source code primarily intended for the CPU. While GPUs and CPUs may have different requirements on algorithms to fully exploit their potential, we restricted ourselves to minor refactoring, even if that meant sacrificing GPU performance in favor of keeping changes to the code as minimal and as encapsulated as possible. Hence, maintainability for the original developers should remain intact. Finally, once the OpenACC implementation was finished we wanted to ensure we could compare its performance to the existing CUDA C port.

In the following, we introduce our methodology to select the features and code paths to begin our endeavor and present the challenges we addressed to create a working in-house OpenACC port of VASP. We show performance measurements of this version and compare it to the same benchmarks when run with the CUDA C based VASP 5.4.4. Finally, we discuss the differences that put the OpenACC-based approach ahead of VASP 5.4.4. We show that the ability to focus more easily on whole application performance, and to quickly port entire call trees, really pays off as a strategy to accelerate scientific codes that pose similar challenges as VASP.

II. USE CASE ANALYSIS

A. *The science and the implementation in VASP*

Scientifically, VASP is used mainly but certainly not exclusively by quantum-chemists, physicists and material scientists to predict properties of mainly solid-state systems, i.e., crystals and surfaces, but also of atoms, molecules and liquids. It does so by calculating their electronic structures as approximate solutions to the many-body Schrödinger equation from first principles, and enables molecular-dynamics calculations on a quantum-mechanical foundation. Many levels of theory and – by that also of accuracy – are implemented into VASP. Among those are the Hartree-Fock (HF) approximation and 2nd-order Møller-Plesset perturbation theory (MP2), Green's functions methods (GW, ACFDT-RPA), time-dependent excitation methods (TDHF and BSE) and density functional theory (DFT). The latter is today's quantum-chemical workhorse and solves sets of the so-called Kohn-Sham (KS) equations

$$H^{KS}\,\psi_n(\boldsymbol{r}) = \varepsilon_n\,\psi_n(\boldsymbol{r}). \qquad (1)$$

Here, H^{KS} is the effective Hamiltonian, while $\psi_n(\boldsymbol{r})$ and ε_n describe the eigenfunctions and -values of the associated KS-orbital n, respectively. Due to the quantum-mechanical foundations, these orbitals must be orthonormal, which is why a significant part of the runtime of a VASP calculation is spent in linear algebra libraries. To describe the periodic wavefunctions, the projector-augmented-wave (PAW) method is chosen. Besides smoothing wavefunctions by a linear transformation, it employs plane waves

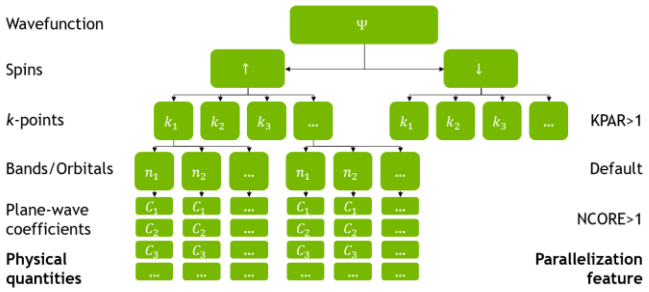

Figure 1. Visualization of the MPI-based layers of parallelization available in VASP 5.4.4. Top level parallelization distributes work over multiple ***k***-points which is controlled by the KPAR keyword. The default distribution scheme is over KS-orbitals and optionally finer-grained distribution can be enabled over the plane-wave coefficients using the NCORE keyword.

$$\psi_n(\boldsymbol{k},\boldsymbol{r}) = \Omega^{-1/2}\,\Sigma_{\boldsymbol{G}}\,C_{\boldsymbol{G},n}(\boldsymbol{k})\,\exp\{i\,(\boldsymbol{k}+\boldsymbol{G})\,\boldsymbol{r}\}. \qquad (2)$$

The coefficients $C_{\boldsymbol{G},n}(\boldsymbol{k})$ are the main quantity that is to be determined in VASP by iterative, self-consistent refinement during which they are repeatedly transformed between Fourier (***k***+***G***) and direct (***r***) space. This results in an important dependency on FFT routines between custom kernels which, for example, apply the Hamiltonian.

VASP has been under active development and refactoring for about 25 years. Most of the code base is written in Fortran 90. On the CPU-side, the current release at the time of writing (5.4.4) has so far been parallelized with MPI only, whereas endeavors toward a hybrid OpenMP/MPI scheme have been reported [7,11]. For distributing the work over the ranks, VASP offers three layers of parallelization of which only one is used by default (see Figure 1 for visualization). The KS-orbitals are mostly independent from each other, but communication is inevitable for the described orthonormalization. Generally, this default scheme offers the best trade-off between load-balancing and data-exchange overhead. On a higher level, larger chunks of the workload can be distributed over ***k***-points, but such a distribution is not always viable, even though communication overhead would also be low, because there are systems that comprise of only a single such point. On the other end, distribution over the plane-wave coefficients is also implemented in VASP. Since this introduces a lot of reductions within the innermost loops, it is crucial to adapt this scheme to the underlying topology. Generally, this distribution scheme is best used only within one (NUMA-) node.

B. *Real-world use cases*

When starting from scratch to accelerate a software package which offers a highly diverse set of features and algorithms, it is obvious that not all functionality can be ported right away. To decide whether a new approach is worth pursuing and potentially even superior over another, it is crucial to begin with use cases that are representative of what users of the application run on a daily basis, while still allowing to compare against features implemented in the

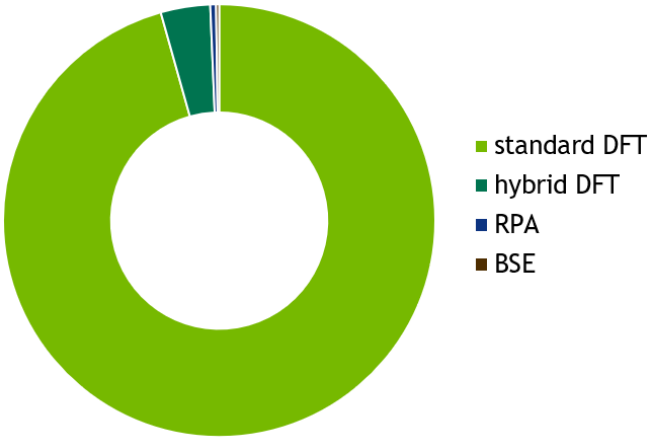

Figure 2: Percentage of quantum-mechanical methods used for VASP jobs in terms of count that ran on Edison late 2014. Standard DFT makes up for 96% of the jobs, but the remaining methods have (heavily) increased computational complexity and machine hours have not been accounted for.

existing port. One option to determine what those use cases are is to poll as many experts in the field as possible, asking what they think is most important. Ultimately, one cannot prevent bias in such opinions based on respective backgrounds and experiences. Another option would be to gather data about jobs run at HPC computing centers. Unfortunately, such statistics are hard to interpret because different users have different needs and they usually stick to only a few supercomputers. This would require averaging over as many computing facilities as possible. However, this would still not include the use cases from on-premises installations, which are not run at major centers.

Keeping that in mind, any hints toward real-world usage are helpful and we were provided with usage data on VASP collected on NERSC's Cray XC30 supercomputer "Edison" in late 2014. Of course, requirements on methodological accuracy might have changed since then and the dataset did not contain data on machine hours, so we can only correlate select features to job count. Hence long running jobs with many nodes have the same weight as short jobs on a few nodes.

In terms of job sizes, the evaluation of the data showed that 34% of all jobs ran within a single node only, while another 28% occupied only 2-4 nodes. In other words: more than half of the submitted jobs were required 4 or less nodes. Assuming a GPU-node would be 4x faster than an unaccelerated node, half of the jobs should fit into a single accelerator node. Even though 95% of all examined jobs were using 12 nodes or less, the remaining 5% may very well make up for a lot more machine hours. To cover most jobs, we focused on smaller node counts first.

In terms of levels of theory, Figure 2 shows that 96% of the jobs rely exclusively on standard DFT. About 4% of the jobs were using hybrid DFT, i.e. to increase overall accuracy, they include results of exact exchange, as defined by the HF method, into the final result. This requires an increased computational complexity and even though machine hours were not part of the dataset, it can be

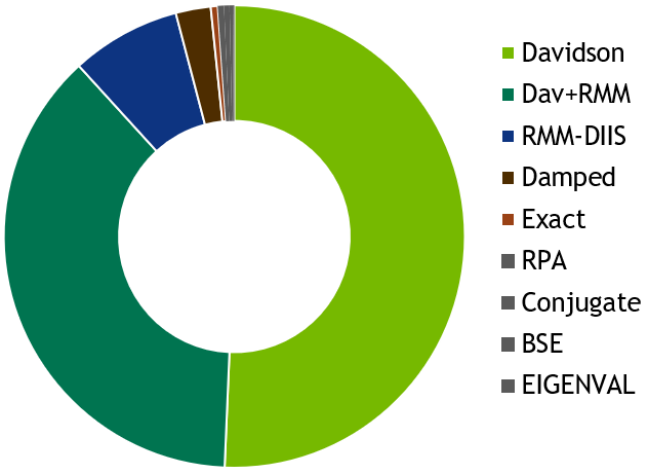

Figure 3: Percentage of main solving algorithms used for VASP jobs in terms of count that ran on Edison late 2014. The blocked Davidson algorithm is default and was used 51% of the cases. The RMM-DIIS scheme on its own was used in 8% of the cases, but is also the predominant step, when combined with Davidson in another 38%. Specialized algorithms share the remaining 3%.

expected that the percentage of hybrid DFT would be considerably higher if machine hours would be the reference metric. RPA and BSE seem not to have played a significant role, but they are also much more computationally demanding than standard DFT, so machine hour percentage could show a different picture here as well. Additionally, both methods are fairly new implementations as compared to DFT and HF, so a more recent statistic could show a larger share for them, as their popularity increases.

Turning back to selecting what levels of theory should be tackled first for the porting effort presented here, Figure 2 gives a clear answer: standard DFT. In relation to the increased complexity, hybrid DFT would be the second in line.

VASP implements several main solvers like the blocked-Davidson or the residual minimization method with direct inversion of the iterative subspace (RMM-DIIS) methods. It does not come as a surprise that the default Davidson algorithm was used in the majority of the jobs, i.e. 51% (see Figure 3). This is probably due to its known good stability, whereas the RMM-DIIS algorithm that is known to be a lot faster makes up for 8%. To alleviate the lesser reliability of RMM-DIIS to directly converge to the desired result, it pays off to combine it with a few steps of blocked-Davidson in the beginning. For the latter combined scheme, RMM-DIIS often is the predominant part, especially for slowly converging calculations. So RMM-DIIS plus the combined scheme sum up to 46% of the jobs. As both algorithms are nearly equally important, the decision on where to start needs to be made on different grounds: we decided to start with RMM-DIIS because it allowed for an easier 1:1 comparison between the performance of the CUDA C and OpenACC ports.

A last option to be discussed is the projector evaluation scheme, because this also leads to different call-trees to be ported first. VASP offers three options: the reciprocal-space,

real-space and automatic scheme. While the reciprocal scheme is the most accurate one, the real-space method offers significant computational advantages when dealing with large supercells. The automatic scheme is also computed in real space and gives the same call-tree within the relevant code parts. Because the job statistics gave a 50:50 distribution, we selected the real-space scheme first for two reasons: a) it helps with larger cells and acceleration sounds more attractive the larger the job is and b) the reciprocal scheme is not supported in VASP 5.4.4 and would not have allowed for comparisons.

The usage statistics have been tremendously helpful to select the features that we focused on supporting first: parallelization schemes seem not of utmost importance at first sight, since most of the examined jobs could maybe even fit into a single accelerator node. Besides, we chose to look at standard DFT with the RMM-DIIS algorithm in the real-space projection scheme first, but blocked-Davidson seems just as important regarding next features to port.

III. PORTING WITH OPENACC

A. *Managing discrete memory*

From the outset, OpenACC was designed to be performance portable across processor architectures. It allows you to write programs that will map efficiently not only to GPUs, but also to multicore CPUs and manycore processors. At compile time it is decided to target multicore CPUs or GPUs by specifying one or more target architectures in the compilation options. When compiling for a target that includes a discrete memory, such as a GPU, managing the latter needs to be addressed somehow.

OpenACC includes data movement directives for managing movement between host memory and accelerator memory. These can safely be ignored for targets that don't need them and the resulting program will still be correct. Besides that, techniques like managed memory are available for NVIDIA GPUs that give a seamless mode of operation by moving data to device memory on demand. While this can offer various advantages of its own, it would also allow for fastest porting, implementing a strategy of dealing with low-level memory management only on an as-needed basis. By design, managed memory can only work with allocated data because an allocation call is replaced by functions that allocate and register the requested memory with the runtime that will then take care of data movement. Statically declared data cannot be moved without help from the operating system, though. Until such functionality becomes widely available there is no way around directives in codes that use such data.

OpenACC data directives for Fortran allow all intrinsic datatypes to be used as arguments. Derived types can be handled in an analogous manner, but need further treatment: when a derived-type variable is copied to the device, OpenACC will transfer the base pointer and all statically defined members. If the derived type contains dynamically allocated data, these need to be transferred separately and after the parent structure has been created on the device. The same holds for other members that are of derived type themselves. In addition to treating their dynamically allocated members and possible further derived types, a derived-type member needs to be adapted on the device because during the initial copy it contained the associated memory location of the host's member. After the latter has been transferred to the device as well, its memory location on the device needs to be used in the device copy of the parent type. That process requires to use the OpenACC pointer attach feature. Of course, when moving such structures back to the host, the process needs to be reversed (detach operation).

For codes that employ deeply nested derived-type data structures like VASP, writing custom control routines that take care of the described "deep-copy" operations was the crucial step to proceed with the port. We wrote routines that handle the deep-copy analogues to the create, delete, copyin, copyout and update OpenACC data directives in either direction, and encapsulated them in a separate module. Since the compiler cannot treat them as directives, their call sites need preprocessor treatment to not interfere with the existing CPU code. Writing these routines seems tedious but is a straightforward process and OpenACC 3.0 will hopefully include fully automatic deep-copy directives that will alleviate this porting overhead.

What is not as straightforward, is to limit the transfers and memory usage on the device to the bare minimum. Moving data back and forth more often than necessary will of course hurt performance. Beyond that, it is well possible that there are parts of complex derived-type structures that could be stored on the device as part of a full deep-copy, but are never referenced within any compute kernel or library call. Using the approach of manual deep-copy as we do here, removing these transfers and freeing device storage is as easy as removing the associated lines from the data-management routines after identifying the opportunities. Proceeding in that fashion sounds like unnecessary overhead, but can lead to quicker success during initial porting. While it can be hard to figure out what data is missing in a complex kernel that crashes for this reason, any redundant data can be removed in a subsequent optimization step.

B. *Compute kernels*

Beyond managing data movement, porting VASP to GPUs with OpenACC mostly consisted of interfacing FFT and linear algebra operations to the respective GPU libraries, cuFFT, cuBLAS and cuSolver, plus adding OpenACC directives to the large number of custom kernels in between. On purpose, we did not refactor loops, e.g., to enable batching for a set of FFTs. This is usually beneficial for GPUs, whereas CPUs often profit more from grouping together multiple operations that work on the same parts of the data because of cache re-use. This compromises GPU performance in favor of a unified code base that helps with maintainability.

Our presented strategies worked out well throughout all parts that we have ported, except for two kernels: one that calculates the non-local contribution of the Hamiltonian with the real-space projection scheme, and another that projects the KS-orbitals onto projection operators within the same

scheme. Structurally, both routines are comprised of multiple nested loops, mainly calculating offsets and pre-factors, and at the innermost level use multiple consecutive loops with a tiny matrix-matrix multiplication (GEMM) in between. Such structures make it hard to expose all the possible parallelism, while not launching excessive amounts of separate tiny kernels. It is technically possible to use device-side cuBLAS functions from within an OpenACC kernel, but without intra-kernel synchronization control that would mean serializing the GEMM, disregarding the possible parallelism and hence dramatically impairing performance. For this reason, we added two routines that calculate the same quantities but structured them to expose as much parallelism as possible. They perform much better on GPUs than the original CPU routines, but are slower than those when compiled without OpenACC. While introducing these two GPU specialized routines appears to deviate from our guideline to not diverge CPU and GPU code, this was a necessary step where lower-level optimization had to be done. In terms of code maintainability, handling only two specialized routines throughout the whole code should still be manageable in contrast to replicated entire sections of the call-tree.

C. *Communication and GPU activity*

At various places data which resides on the GPU needs to be exchanged between MPI ranks. With OpenACC 2.6 and with a GPU-aware MPI implementation, this is as easy as enclosing every MPI call with the HOST_DATA construct. By adding the IF_PRESENT clause, attempts to pass a device pointer are only made when the data really resides on the GPU. This is important as certain subroutines containing the communication can be shared between a calling routine that runs on the GPU and another one that does not. Nevertheless, care should be taken with communications, when data to be communicated is also needed on the CPU. In this case, it is a matter of the algorithm, communication pattern and also interconnect topology, if updating the data on the host and initiating the communication from there can perform better than the other way around.

To deal with (de-)activation of GPU functionality, we found it beneficial to introduce an on-off-switch control infrastructure that is respected by every kernel and by every call to a library function. This eases debugging tremendously and facilitates sharing of routines that contain ported kernels between GPU and CPU.

IV. PERFORMANCE COMPARISON

A. *Whole application*

So far, we have discussed which challenges needed to be addressed for an OpenACC port of VASP, which adheres to the guidelines outlined in the introduction. While we have already mentioned that many optimization strategies have purposefully not been applied, the obvious concern and ultimate question to be answered is, how well the resulting OpenACC port performs on real-word benchmarks as discussed in section 2. Hence we compare the new OpenACC port to the CUDA C based port as available in

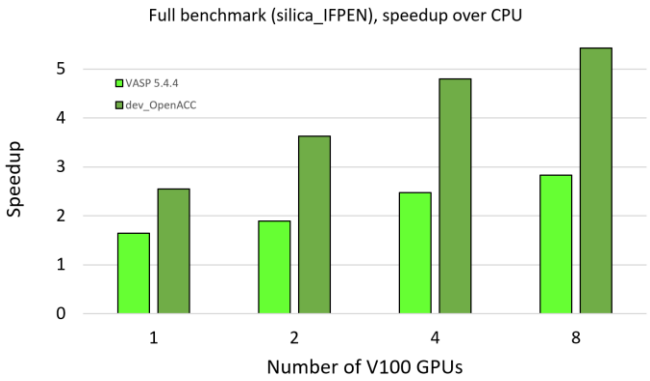

Figure 4: Comparison of the speedups over a CPU-only run between the CUDA C based GPU accelerated port of VASP 5.4.4 and our new OpenACC based porting effort for 1 to 8 Tesla V100 accelerator cards for the silica_IFPEN benchmark, which features the RMM-DIIS algorithm.

VASP 5.4.4. All benchmarks presented in the following were run on an NVIDIA DGX-1 system that hosts two Intel Xeon E5-2698 v4 Broadwell CPUs that are accompanied by 8 NVIDIA Tesla V100 GPUs. To ensure fair performance comparison in all tested cases, all runs based on VASP 5.4.4 were using Intel's compiler suite, MKL and their implementation of MPI. The CPU-only runs always used all 40 available cores.

The discussed NCORE parallelization scheme can help with efficiency for CPU runs. Our benchmarks stay within a single node, so CPU-wise we achieved best performance when matching NCORE with the number of available CPU cores, i.e. 40. For GPU runs, NCORE needs be set to 1. As a side-effect the CPU-only calculations have indeed a smaller workload than the GPU calculations. When distributing the plane-wave coefficients over all MPI ranks, no workload distribution will be carried out over the KS-orbitals (default scheme, see Figure 1). Consequently, there is no need to increase the number of calculated KS-orbitals to a value that is divisible by the number of ranks, which easily happens for NCORE = 1 calculations. Nevertheless, comparing such different calculations is reasonable: adding some KS-orbitals will not alter the scientific result. Thus, speedups between the calculations presented here are time-to-solution comparisons against an optimized setup for the CPU-only runs. Other runtime parameters like NSIM (that controls how many orbitals are treated in a block) have been tuned for best performance with the respective architecture and port.

1) *RMM-DIIS*

To measure and compare the performance of the OpenACC-based approach we chose the silica_IFPEN benchmark as a test dataset. On the one hand, it matches our previously discussed choices in term of level of theory, solver and projector evaluation scheme. On the other hand, it has been prominent throughout the development of the CUDA C port and we want to compare our efforts against datasets for which the CUDA port can be expected to have been well tuned. The CPU-only runtime of the complete calculation took 465 s.

Figure 4 compares the speedups of the GPU runs with VASP 5.4.4 and the OpenACC port over CPU-only 5.4.4

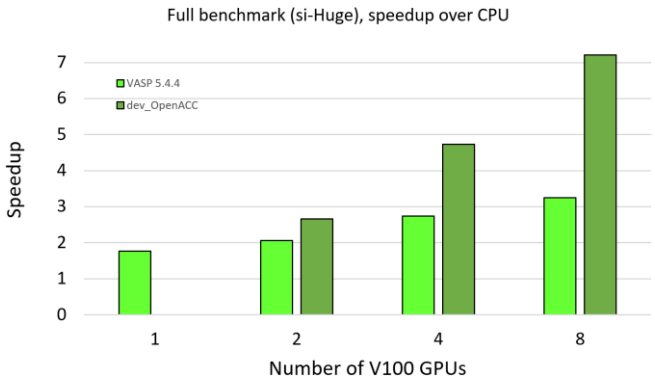

Figure 5: Comparison of the speedups over a CPU-only run between the CUDA C based GPU accelerated port of VASP 5.4.4 and our here presented OpenACC based porting effort for 1 to 8 Tesla V100 accelerator cards for the si-Huge benchmark that features the blocked-Davidson algorithm.

runs. Version 5.4.4 showed maximum performance when run with multiple ranks per GPU, i.e. 8, 4, 2 and 1 rank per GPU for 1,2,4 and 8 GPUs, respectively. To achieve this, the NVIDIA CUDA Multi Process Service (MPS) was employed. The same technique helps with OpenACC as well, but it turns out fewer ranks are needed for best performance, i.e. 4, 2, 1 and 1 ranks per GPU, respectively. Using MPS introduces some start-up overhead which can become significant for benchmarks that finish in only 86 s on 8 GPUs: disregarding the initialization phase, 2 ranks per GPU outperform 1 rank per GPU also for 4 and 8 devices.

It may come as a surprise that the OpenACC approach indeed outperforms 5.4.4 for any tested configuration. To not open a discussion on OpenACC versus CUDA execution speeds here, we stress that the share of accelerated routines between both approaches is not identical. But even though we did not apply aggressive optimizations, our result outperforms the existing port notably. The reason is probably that the porting strategy coming along with OpenACC, allowed us to focus on whole application performance and to quickly be able to accelerate some regions as well, which remained on the CPU in the 5.4.4 release.

2) Blocked-Davidson

As a second testcase we examined the performance of a benchmark that uses the blocked-Davidson algorithm. The same arguments presented to select silica_IFPEN, apply to the si-Huge benchmark as well. A notable difference besides the main solver is that it is a bigger system, which translates to its CPU-only execution time of 4852 s. This would alleviate penalties paid during initialization, but nevertheless best performance for GPU-accelerated 5.4.4 was achieved with 4, 2, 2 and 1 ranks per GPU, which suggests that MPS apparently hinders GPU execution speeds for this dataset but is beneficial to increase CPU processing speeds by enabling more cores. For the OpenACC port, no multiple ranks per GPU configuration performed better than a direct mapping, excluding the single GPU setup. The latter could not be run with the OpenACC port, because of insufficient device memory on a single 16 GB V100, as in its current development state, the port uses more device memory than 5.4.4.

The si-Huge benchmark shows an even higher gap between 5.4.4 and our OpenACC port. But even though there is still room left for optimization, the scaling efficiency to more accelerator cards has dramatically improved: For 8 GPUs the new approach is more than 2x faster than the existing accelerated version. Yet again, we argue that this is not caused by OpenACC itself, but more based on the freedom it offers to focus on the right strategy for highly complex code bases.

B. Kernel-level comparison

To further analyze where the performance advantages stem from, we switch from full benchmark comparisons to the finer level of individual kernels. This is not as trivial as it may sound because kernels cannot be mapped 1:1 between the two ports. One reason is that in the CUDA C based approach, heavy refactorings have been applied. Fusing kernels that traverse the same memory is a worthwhile optimization used in the CUDA port. But we refrained from using it in the OpenACC port. Starting a single kernel instead of multiple ones can reduce launch latencies, but depending on code structure, yet again, would require refactoring.

The challenge was to identify a set of kernels in either port which calculate the same quantities and only those. Table 1 lists the runtimes of kernels that calculate the energy expectation values for a block of NSIM = 4 KS-orbitals within the silica_IFPEN benchmark, which basically means doing NSIM independent reductions. The primary difference between both approaches is that the CUDA C port launches one kernel per orbital and after all of them are finished, another kernel per NSIM-block. The OpenACC port just launches 8 kernels per orbital without a fused post-processing kernel. To properly fuse the post-processing, a synchronization point is inevitable. We can only guess that the implementation in the CUDA C port was beneficial on earlier GPU generations. OpenACC can adapt its optimizations according to the architecture at compile time and in this case results in a 15% faster code, even though it does not make use of aggressive optimization techniques.

C. Section-level comparisons

A main solving algorithm in VASP can be grouped into various high-level sections. Depending on that section, the share of subroutines that run on the GPU can vary drastically between the CUDA C approach and the OpenACC port. To directly compare CUDA C and OpenACC performance one should examine a section that has the least amount of

Table 1: Kernel level comparison between OpenACC and CUDA C based approaches for energy eigenvalue optimization

	CUDA-C port	***OpenACC port***
Kernels per orbital	1 (69 µs)	8 (90 µs total)
kernels per NSIM-block (4 orbitals)	1 (137 µs)	0 (0 µs)
runtime per orbital	103 µs	90 µs
runtime per NSIM-block (4 orbitals)	413 µs	360 µs

differences which are unrelated to these two frameworks. Before we do so, let's look at a section that differs widely first, still sticking to the silica_IFPEN dataset:

1) Orthonormalization

In Table 2 we give runtimes of subsections needed to orthonormalize the orbitals. The first step is to redistribute data, which in the CUDA C port requires transferring the data back to the host first. The OpenACC port in contrast saves 75 ms by saving these buffer copies and in addition can benefit from lower latency and higher bandwidth GPU-GPU NVLink-interconnect as available on the employed DGX-1 platform. Of course, the exchanged data needs to be brought back to the GPU for the subsequent GEMMs, that is not as costly as in the first step because the CUDA C port uses streaming to hide the transfer behind computation and loses only 4 ms. Saving a comparably small memory transfer and probably designed prior to the latest performance improvements in cuSolver, 5.4.4 kept the Cholesky decomposition on the CPU, which nowadays loses another 12 ms. For further GEMMs we sacrificed device memory consumption in OpenACC for performance, saving another 17 ms and the final MPI data exchange sets the CUDA-C port back for another 105 ms. While GPU-aware MPI gains roughly 40% in this section, another 7.5% have been saved by a more complete port of the section.

2) EDDRMM

Finally, we turn to a section where influences by different MPI implementations and additionally ported subsections and -routines are minimal. Yet, different implementations as exemplified in Section IV. B show their effect. Figure 6 shows that for the EDDRMM section for single GPU runs, VASP 5.4.4 achieves comparable performance as the OpenACC port. But for 8 GPUs, the new port has a clear advantage due to improved scaling. For both approaches using MPS to increase the number of MPI ranks mapped to a GPU is beneficial, within this section. While EDDRMM alone cannot explain the improvement over the existing port in 5.4.4, it clearly plays a major role given it is one of the most time-consuming sections, taking 17% of total runtime for 8 GPUs.

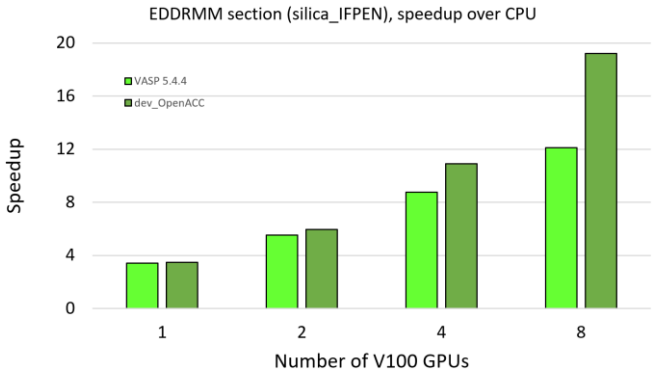

Figure 6: Comparison of the speedups over a CPU-only run between the CUDA C based GPU accelerated port of VASP 5.4.4 and our here presented OpenACC based porting effort within the EDDRMM section only for 1 to 8 Tesla V100 accelerator cards for the silica_IFPEN benchmark that features the RMM-DIIS algorithm.

V. CONCLUSION

We have shown that OpenACC enables straightforward porting of complex codes, after managing device memory has been addressed with techniques such as manual deep-copy. Aggressive optimizations that lead to major code refactoring were not applied on purpose and only two kernels needed to be rewritten to express more parallelism. Focusing on whole application performance has paid off because our first attempts on the RMM-DIIS and blocked-Davidson solvers outperforms the existing CUDA C port in VASP 5.4.4 for all benchmarks that we have tested by a significant margin, while the applied strategy suggest that there is still room for further optimization.

Once presented with our preliminary performance numbers and after they were able to compare the ported source code to the original CPU version, the developers of VASP at the University of Vienna have decided that OpenACC is the way to move forward to accelerate VASP with GPUs.

ACKNOWLEDGMENT

We thank Zhengji Zhao and everyone at NERSC who helped gathering the usage statistics on VASP in their facility for providing the data and the fruitful, associated discussion. We are also grateful to Georg Kresse and Martijn Marsman for their continued interest in the project and helpful, open discussions around it.

REFERENCES

[1] G. Kresse and J. Hafner, "Ab initio molecular dynamics for liquid metals", Phys. Rev. B, vol. 6, pp. 558−561, January 1993.

[2] G. Kresse, and J. Hafner, "Ab initio molecular-dynamics simulation of the liquid-metal–amorphous-semiconductor transition in germanium," Phys. Rev. B, vol. 49, pp. 14251−14269, May 1994.

[3] G. Kresse, and J. Hafner, "Norm-conserving and ultrasoft pseudopotentials for first-row and transition elements," J. Phys.: Condens. Matt., vol. 6, pp. 8245, 1994.

[4] G. Kresse, and J. Furthmüller, "Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set," Comput. Mat. Sci., vol. 6, pp. 15–50, July 1996.

[5] G. Kresse, and J. Furthmüller, "Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set," Phys. Rev. B, vol. 54, pp. 11169−11186, October 1996.

[6] G. Kresse and D. Joubert, "From ultrasoft pseudopotentials to the projector augmented-wave method," Phys. Rev. B., vol. 59, pp. 1758−1775, January 1999.

[7] Z. Zhao, M. Marsman, F. Wende, and J. Kim, "Performance of

Table 2: Section-level comparison between OpenACC and CUDA C based approaches for orthonormalization of the KS-orbitals

	CUDA-C port	***OpenACC port***
Redistributing wavefunctions	Host-only MPI (185 ms)	GPU-aware MPI (110 ms)
Matrix-Matrix-Muls	Streamed data (19 ms)	GPU local data (15 ms)
Cholesky decomposition	CPU-only (24 ms)	cuSolver (12 ms)
Matrix-Matrix-Muls	Default scheme (30 ms)	better blocking (13 ms)
Redistributing wavefunctions	Host-only MPI (185 ms)	GPU-aware MPI (80 ms)

Hybrid MPI/OpenMP VASP on Cray XC40 Based on Intel Knights Landing Many Integrated Core Architecture", In: CUG Proceedings, 2017.

[8] S. Maintz, B. Eck, R. Dronskowski, "Speeding up plane-wave electronic-structure calculations using graphics-processing units," Comput. Phys. Commun., vol. 182, pp. 1421−1427, July 2011.

[9] M. Hutchinson, M. Widom, "VASP on a GPU: Application to exact-exchange calculations of the stability of elemental boron," Comput. Phys. Commun., vol. 183, pp. 1422−1426, July 2012.

[10] M. Hacene, A. Anciaux-Sedrakian, X. Rozanska, D. Klahr, T. Guignon, P. Fleurat-Lessard, "Accelerating VASP electronic structure calculations using graphic processing units," J. Comput. Chem., vol. 33, pp. 2581−2589, December 2012.

[11] F. Wende, M. Marsman, Z. Zhao, J. Kim, "Porting VASP from MPI to MPI+OpenMP [SIMD]", In: IWOMP 2017, pp. 107−122.