OpenACC - a qualitative study

Jonas Ernst

May 2022

Abstract

OpenACC is a programming standard which lets the programmer tell the compiler where to parallelize the code. This approach tries to make speed-ups using GPUs easier while maintaining the same speed than native GPU code. In this paper we will have a look at different projects using OpenACC and analyse them in terms of performance gain. We conclude that these speed-ups can range between two to ten times over CPU-only code depending on the systems used. Compared to non-portable solutions OpenACC manages to keep roughly 80% of the performance.

1 Introduction

In recent years the single thread performance of processors (CPU) was not improving at the rate that it used to. Also the trends in high performance computing (HPC) shifted from homogenous architectures to heterogeneous systems with highly parallel accelerator cards. For these newer systems good parallelised applications are needed in order to utilize the full computing power of those systems. However, developing directly for these architectures requires a lot of manpower, is more confusing for the developer [8] and depends on the system you are working on and therefore is not very portable.
OpenACC is a programming standard developed by Cray, CAPS, Nvidia and PGI and it was announced in 2011. The standard tries to make porting code to a graphics- or accelerator card (GPU) easier for the developer by letting the compiler do most of the heavy lifting. The programmer just annotates the source code where the compiler should offload the work to the GPU by using compiler directives and API-calls. This approach promised ease of use and with that also implementation speed. The standard is defined for C, C++ and Fortran and the newest release at the time of writing is version 3.2 from November 2021. However, most compilers do not support the newest one. For example, supported by GGC 10 and 11 is version 2.6.
In the following pages 3 different projects that use OpenACC to offload work to the GPU will be shown. These are then analysed what effect this programming standard can achieve in terms of speed-up compared to CPU-only systems. At last the portability of OpenACC across different HPC systems will also be taken into consideration, while comparing OpenACC to similar projects like OpenMP.

2 Projects

2.1 Many-Fermion Dynamics-nuclear

The First Project is a configuration interaction code [3, 6] that is used for nuclear structure calculations, so it approximates the many-body wave function of self-bound atomic nuclei starting with two- or three-nucleon interactions. For this paper it not the physics behind it are not relevant. For understanding the computation however, it is important to know that is the solutions to the problem statement are a "few (five to ten) low lying eigenpairs" of a large sparse symmetric matrix. ([6], p. 2) In iterative implementations this would be realised using sparse matrix vector multiplications. A parallel version for this already exists that uses the "Locally Optimal Block Preconditioned Conjugate Gradient method"

(LOBPCG). This method relies on multiplication of a sparse square matrix on a tall skinny matrix (SpMM). This has the effect of better concurrency and effective approximations to eigenvectors. It also allows for a preconditioner phase which can be used to accelerate convergance. The Code for this was written in Fortran 90 with parallelisation consisting of a mix of message passing interface (MPI) and OpenMP. MPI is used to send messages between the different tasks and processes, while OpenMP is a project similar to OpenACC. It enables easy multiprocessing on shared-memory devices e.g. multi-core CPU's by using compiler directives. However the a major difference is that OpenMP allows the developer to specify how to parallelize directly(e.g. number of threads). On the other hand in OpenACC this is done by the compiler, which keeps easy portability accross different systems. [8]
This code was then ported to GPU accelerated systems. OpenAcc was chosen as it keeps flexibility of running on multi-core CPUs. Furthermore the project keeps its portability across different architectures. The communication between different processes still relies on MPI. The implementation of SpMM was like the original OpenMP version due to their similarities and the directives could just be replaced. SpMM still needed a few changes because the two programming standards do not always support the same functionality. There were also some changes made to the implementation to better leverage the parallel computing power of GPUs. For example, with enough memory it is easy for CPUs to use private arrays and keep good cache locality and therefore gain superior performance. The GPUs on the other hand are much more constrained by memory available per worker thread due to the order of magnitude more in parallelism. Private arrays on the GPU limit the amount of worker threads one can use due to memory limitations and thus do not harness the whole power of the GPU. Instead, the worker threads use the shared memory pool and index offsets.
The speed-up of this port was assessed on three different systems. (1)

	Cori GPU	Cori GDX	Spock
GPU	NVIDIA V100	NVIDIA A100	NVIDIA MI100
CPU	Intel Sky-lake	AMD Rome	AMD Rome
GPUs/Node	8	8	4
Bus	PCIe 3.0	PCIe 4.0	PCIe 4.0
per GPU Memory	16GB	40 GB	32 GB

Table 1: System tested Many-Fermion Dynamics-nuclear on ([3] Table 1)

Vendor	Version	OpenACC	OpenMP	V100	A100	MI100
NVIDIA	21.7	201711 (2.6)	202011 (5.1)	✓	✓	
HPE/ Cray	12.0.1	201306 (2.0)	201511 (4.5)	✓		✓

Table 2: Compilers used for project 1 ([3] Table 2)

As you can see in Fig. 1 and Fig. 2 when it works the OpenACC implementation can beat even the best OpenMP implementation. In Fig. 1 [6] we can see that the Eigensolver can run up to 7 times faster on the GPU than on the CPU. More impressively is the speedup during SpMM [6]. The GPU version can achieve a speed boost of 15 to 20 times in raw computing time and 9 to 2 times with communications and data transfers introduced by the MPI. This difference in numbers is explained by the larger memory requirements of a larger matrix. Because the larger the Matrix gets, the more nodes have to get used which leads to more resource sharing and a larger communication overhead. Also, the time doing the dense linear algebra calculations on the GPU is almost not existent compared to the time spent in SpMM and the precondition phase. These tests can utilise the GPUs processors around 50% of their theoretical raw computing power (in FLOP/s) and the memory around 80%. ([6] Fig. 9)

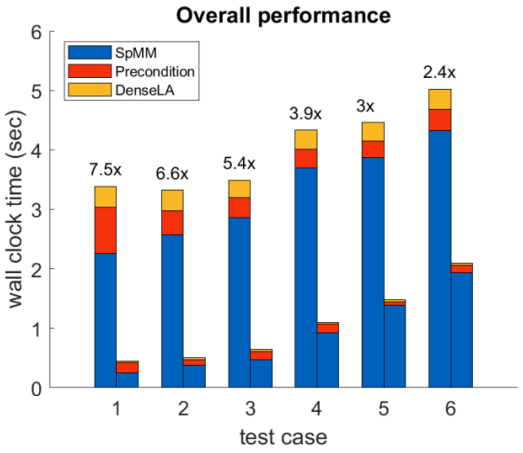


Figure 1: Comparison between the overall performance of LOBPCG on GPU with that on the CPU, taken from [6] p. 23

In Figure 2 it can be observed that array reduction in OpenACC matches the performance of OpenMP or is slightly worse. Nonetheless it is still preferable to calculate these on the GPU, because otherwise one would have to move data off the GPU to the CPU to process them there. This is slow however and should generally be avoided.
The developers [3] also state that the use of OpenACC is good for porting OpenMP-code to GPUs, but still needs some knowledge of the different technologies and how to best use them to get the best results. This alignes with statement from [5]. To use OpenAcc 1500 additional words of code (WOC) had been used. For a native CUDA version 13000 additional WOCs were needed.
For large matrices, the total work also must be shared more between the different computing nodes. Therefore the effective gain in speed is limited by communication speed between the nodes (MPI).

2.2 Nek5000

The next project is called Nek5000 [9]. It is used for high-fidelity Computational Fluid Dynamics (CFD) which is needed to study the air around airplane

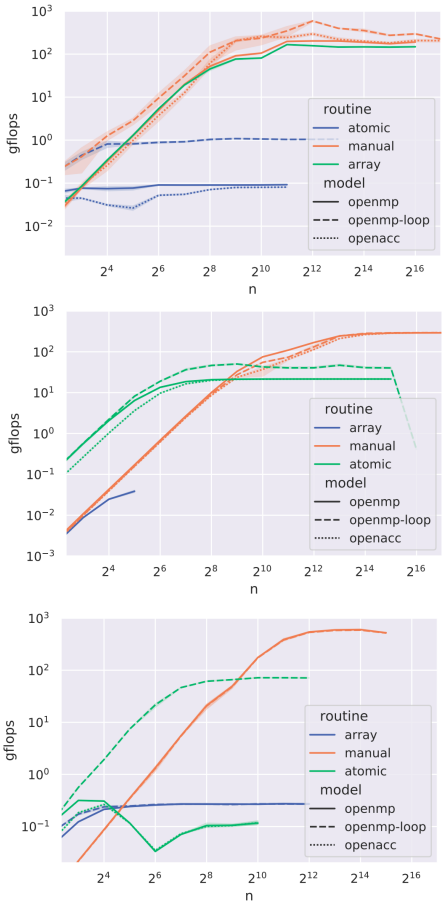


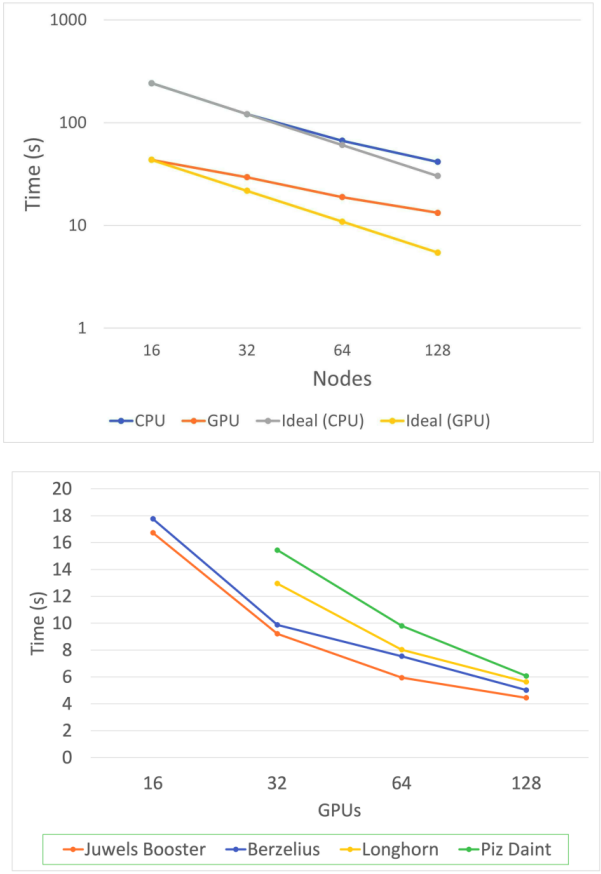
Figure 2: "Performance of array reduction with array size of 64 (higher is better). Top: On Skylake CPUs; Middle: On A100, where we encountered run time errors for $n > 2^5$ with OpenMP array reduction and a compile error with OpenMP with loops; Bottom: On MI100, where for $mn^2 2 \geq 2\ 32$ there appears to be a correctness error due to integer overflow on the collapsed loops with the Cray compiler.", taken from [3] p. 17

wings, ocean currents etc. It uses the spectral element method (SEM) which partitions the simulation Volume into "non-overlapping, body conforming hexahedral subdomains called elements". ([9] p. 3) These are then represented by a tensor product of "one-dimensional Lagrange interpolation polynomials." ([9]) The Nek5000 package is regarded as one of the most used HPC frameworks in academic use since it is development in the 1980s. It's written in Fortran 77 and has relied thus far on the MPI for parallelism. This worked so far very good for CPU calculations. However, with the current trends in HPC going away from homogeneous general-purpose processors to more specialised hardware the project needed to undergo some changes to be future proof.
The authors [9] acknowledge that there are already some other projects that do the same thing on modern hardware, but they lack the portability that the authors want.
During the port, the developers noted a few things. Small maths kernels are called repeatedly in a loop. To speed up the code the loop got pushed into these kernel calls. This increases the code complexity, because now these kernels are specific to each loop. Calculation with reductions (like in Figure 2) are quite inefficient and can be replaced by a handwritten kernel (native port) to increase speed. At last, OpenACC seems to be very conservative with moving data around. [9]
The authors [9] state that this algorithm is still memory bound and therefore needs to be very cache efficient to get the most speed out of the system. The maximum size of different elements which you can efficiently emulate is directly linked to the capacy of available shared memory.
To compare the original implementation to the new version different examples have been simulated. Re_τ is the friction Reynolds number and the maximum polynomial order refers to the Langrage polynom. The speed benefit by using the GPUs on these systems can vary between from 3 to 5 times which can be seen in Figure 3 in the top picture. The difference between CPU and GPU gets smaller the more nodes one enables. This effect can be explained by the decrease in FLOP/s when each node has less elements to calculate.

	Piz Daint	Long-horn	JEWELS	Bezelius
GPU	NVIDIA P100	4x NVIDIA V100	4x NVIDIA A100	8x NVIDIA A100
CPU	Intel Xeon E5-2690 v3	2x IBM Power 9	2x AMD EPYC 7402	2x AMD EPYC 7742
Topo-logy	dragon-fly	spine and leaf	dragon-fly+	fat tree

Table 3: System tested Nek5000 on ([9] p. 5)

The communication between the nodes seems to be a factor too. However, the correlation between number of elements simulated on a Node and performance hinders strong GPU scaling the most. Nonetheless this execution is still faster on the GPUs than it is on the CPUs.
In the bottom diagram we can spot the difference in calculation time of the different testing systems. Note that the first two systems do not have the memory requirements for 16 nodes and thus are not available for testing.
The difference between the two A100 systems derives mainly from the better network architecture and increases the more nodes you enable. The performance gain switching from Jewels to Piz Daint is a factor of roughly 1.5 to 2. This is roughly the factor the of FLOP/s and memory bandwith.
The authors [9] of the original paper conclude that the increase in speed is less than expected by using GPUs rather than CPUs because of the huge increase in raw computing power. This is hindered by two factors. First there is still a CPU only task that can not be done in parallel thus far due to its complexity. This however is hoped to be solved in the future. Secondly moving data between CPU and GPU should be reduced even more. This paper concludes with the statement that the port was a success and Nek5000 can now be used with modern GPUs. The port shows great performance improvements in speed from 3 to 5 times per node. The main limiting factor with a larger number of nodes is the part that isn't easily parallelizable, neither on CPU's, nor on GPU's.



(a)

Figure 3: top: "CPU and GPU performance for $Re_\tau = 550$ with maximum polynomial order $N = 9$, on JUWELS Booster. Ideal scaling based on the 16 node results is also shown.;" bottom: "Results for Reynolds number $RE_\tau = 360$ and maximum polynomial order of $N = 7$", taken from [9] p. 7

2.3 FluTAS

With more HPC servers upgrading to CPU-GPU systems or even GPU-only systems the developers of FluTAS saw the need for a modernised version of an Navier-Stokes-equation solver. The Navier-Stockes-equation model the interaction between fluids and gas. So, this could be modelling a cloud, bubbles in the water etc. ([4] p.2-3). There were already some open-source projects for single phase codes, however for multiphase systems (multiple fluids interaction with each other) there was no project that utilised the GPUs of these systems to better use the new HPC servers. Multiphase flows model the interaction between different fluids. This is were FluTAS [4] came in. FluTAS is a multiphase flow solver written in Fortran 90 and can be run on both CPUs and GPUs. The CPU implementation relies on MPI for its parallelisation while the GPU implementation uses OpenACC.

For the performance test FluTAS was evaluated at MeluXina [2] in Luxembourg and Berzelius [1] in Sweden. The CPU tests on the other hand were performed on the Tetralith in Sweden.

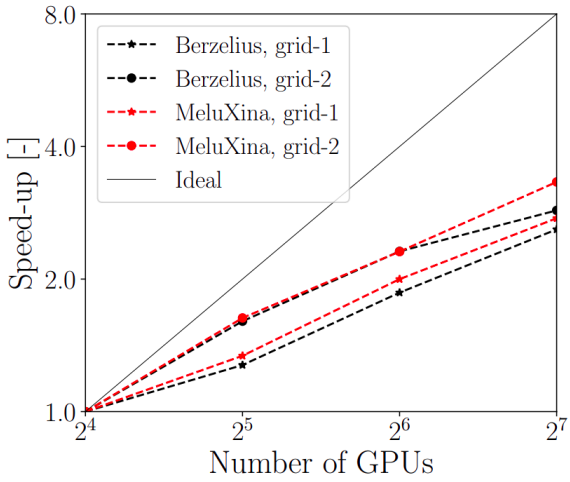


Figure 4: "Strong scaling test performed on Berzelius (black-dashed lines) and MeluXina (red-dashed lines) clusters for two different grids: $1024 \times 512 \times 1024$ (grid-1) and $1024 \times 1024 \times 1024$ (grid-2). The black continuous line indicates the ideal behavior desired for the strong-scaling test.", taken from [4] p.22

Figure 4 shows the speed-up one can achieve with using more GPUs. There is always a net benefit of using more GPUs, but that benefit gets progressively worse the more accelerators are already in the system. This effect is seen less with the test of grid-2

compared to grid-1 because grid-2 is bigger and thus one GPU can be better utilised. The speed reduction compared to the ideal line can explained with a communication increase between the GPUs. More GPUs also lead to a "reduction in problem size on an individual graphics card, which does not leverage the full compute capacity of each GPU". ([4], p. 21)
The paper concludes it is preferable to use newer systems with 80 GB of memory on each card and NVLINK support, a NVIDIA High Bandwidth Bridge between individual GPUs, to lower scaling issues due to communication overhead.

The speed of FluTAS was also compared between

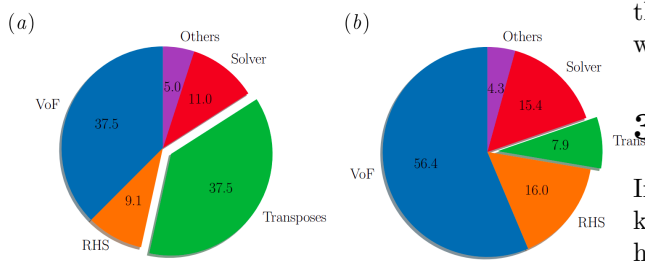


Figure 5: "Comparison of code-section load percentage on the total simulation time for GPUs (panel a) and CPUs (panel b). The different "slices" represent different code sections: 1) VoF (i.e. interface reconstruction and advection, update of the thermophysical properties), 2) RHS (i.e. discretization of the governing equations), 3) Transposes (i.e. transpose operation in the solver), 4) Solver (i.e. only Gaussian elimination) and others (i.e. correction step, divergence/time-step checks, output and post-processing routines).", taken from [4] p. 23

a CPU system and a GPU system. One should note however that comparisons between these are quite complex and there is no standard way. Running these tests on the same hardware (CPU-GPU system) should also be avoided due to the difference in networking architecture of a CPU to a CPU-GPU system. Otherwise, the conclusions might be different what you can expect on native CPU systems.
The weak scaling tests are linear when everything happens within a single node. However, when more nodes are required, slower communications must be used which leads to less of a performance increase. These tests took $0.191s$ on 8 GPUs vs $1.075s$ on 512 CPUs which means that one GPU is equivalent in performance to roughly 359 CPUs according to these tests.
Figure 5 shows the difference in architecture. The CPU System spends most of its time doing for-loops which can be done so much faster in parallel on an accelerator card. The GPU system is mostly hindered by doing transposes which can not be better parallelised. The time it takes for both systems stays roughly the same during these and therefore most of the speed-up is gained in the other parts of the code which use the GPU more effectively.

3 Portability

In this section we will take a look on how OpenAcc keeps its performance across different systems. This has already been intensively studied and now we are presenting one of these studies. [7]. To measure the portability the study defines:

$$\Phi_M = \frac{\sum_{i \in T} e_i(a,b,c)}{|T|} \text{ with}$$

$$e_i(a,b,c) = \frac{\text{portable model e.g. OpenACC in FLOP/s}}{\text{non-portable model e.g. CUDA in FLOP/s}}$$

M is here the Programming model with case studies T. e is the efficiency of a case study of application a solving problem b on system c.

Model	Case Studies	Φ_M	std. dev.	max.	min
OpenACC	109	81%	13%	100%	51%
OpenMP	62	81%	14%	100%	52%

Table 4: Performance Portability, taken from [7] p. 111

As we can see in Table 4 the performance portability is quite similar to that of OpenMP. Both keep

on average around 81% of the speed compared to the non-portable version.

4 Conclusion

The Projects before show promise using OpenACC in porting CPU-only code to GPU-accelerated ones. It's easy for already developed code to be adapted for GPUs by compiler directives, especially porting code from already implemented OpenMP code due to the similar approach in terms of implementation.
The generated code is portable between various architectures when no compiler issues arise.
However, it is no magic tool. One still needs to understand the difference in architecture between GPU and CPU systems and optimise accordingly to get the best benefits. OpenACC shows great promise in optimizing huge for-loops e.g. matrix operations due to the huge potential for parallelisation. Nonetheless OpenACC has also the same limitations a native port to CUDA etc. would have and one still must rely on the system architects to limit memory bottlenecks and/or communication overhead.

References

[1] Berzelius. `https://www.nsc.liu.se/systems/berzelius/`, (accessed July 1, 2022).

[2] Meluxina. `https://luxprovide.lu/technical-structure/`, (accessed July 1, 2022).

[3] Brandon Cook, Patrick J. Fasano, Pieter Maris, Chao Yang, and Dossay Oryspayev. Accelerating quantum many-body configuration interaction with directives. *CoRR*, abs/2110.10765, 2021.

[4] Marco Crialesi-Esposito, Nicolo Scapin, Andreas D. Demou, Marco Edoardo Rosti, Pedro Costa, Filippo Spiga, and Luca Brandt. Flutas: A gpu-accelerated finite difference code for multiphase flows, 2022.

[5] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating hydrocodes with openacc, opencl and cuda. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 465–471, 2012.

[6] Pieter Maris, Chao Yang, Dossay Oryspayev, and Brandon Cook. Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on gpus using openacc. *CoRR*, abs/2109.00485, 2021.

[7] Ami Marowka. On the performance portability of openacc, openmp, kokkos and raja. In *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, page 103–114, New York, NY, USA, 2022. Association for Computing Machinery.

[8] R. Usha, Prachi Pandey, and N. Mangala. A comprehensive comparison and analysis of openacc and openmp 4.5 for nvidia gpus. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2020.

[9] Jonathan Vincent, Jing Gong, Martin Karp, Adam Peplinski, Niclas Jansson, Artur Podobas, Andreas Jocksch, Jie Yao, Fazle Hussain, Stefano Markidis, Matts Karlsson, Dirk Pleiter, Erwin Laure, and Philipp Schlatter. Strong scaling of openacc enabled nek5000 on several GPU based HPC systems. *CoRR*, abs/2109.03592, 2021.