Many–Core Sustainability by Pragma Directives

Andreas Kucher[1] and Gundolf Haase[1]

Institute for Mathematics and Scientific Computing, University of Graz**

Abstract. Many–core hardware is well adopted in scientific computing for a number of applications in an academic setting. Uncertainty about upcoming architectures and large development times for this hardware result in a modest acceptance in industry for commercial use. An upcoming turn from language–based many–core programming towards directive–based frameworks, similar to OpenMP, is an attempt to tackle these issues.
We present a case study for a many–core acceleration of a large–scale commercial CFD solver by means of such frameworks. We achieved a local acceleration of up to 45 for hot spots with recent hardware but the global speedup remains below 2. The main obstacle for an efficient instrumentation is the design and the complexity of the original software. Further, restrictions given by the hardware and the frameworks exist. Based on the results we sketch a long term plan for a further acceleration.

Keywords: computational fluid dynamics, general purpose GPU, many–core, parallelization, OpenACC, OpenHMPP, accelerator frameworks

1 Introduction

Many–core processors such as GPUs can outperform recent CPUs with respect to their compute power in a fine–grained parallel setting at the cost of a higher total power consumption. However, for a successful application of many–core hardware in numerical software, algorithms have to be modified and then implemented in a distinct programming language like CUDA or OpenCL. This has been done for a number of algorithms and the results are available as libraries giving good or excellent performance (e.g. [1]). Without such libraries, a many–core migration of algorithms and their implementation can be challenging.

This results in a good acceptance of many–core hardware in small and research codes for certain applications, but in industry the acceptance is modest. The migration of code to many–core hardware has to be profitable, i.e., the performance gain must be high and the software has to be sustainable at a long term. The impact on CPU performance due to many–core support should be negligible and a many–core migration must not affect regular code enhancements,

** The final publication can be found in I. Lirkov, S. Margenov, and J. Wasniewski, editors, Large Scale Scientific Computing LSSC13, volume 8353 of Lecture Notes in Computer Science, pages 426433. Springer, 2014 and it is available at http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-29842-4

particularly for large groups of developers. Independence of the hardware used and bit-compatibility to CPU results are of advantage.

There is a trend to move from language based many–core programming (CUDA, OpenCL) towards directive–based frameworks (OpenACC, OpenHMPP) similar to OpenMP. This paradigm shift is supposed to reduce development time and allows programmers to invoke many–core hardware without having deep knowledge on either hardware or language. The frameworks are expected to make many–core hardware interesting for commercial use, as they meet the requirements listed above in parts.

In a case study we investigate the applicability and sustainability of directive–based frameworks for a many–core acceleration of a commercial large–scale structured flow solver for CFD simulations. Flux computations on GPUs have been investigated in [2]. The potential of GPUs for CFD with respect to performance has been proven in [3] if the code can be fully matched with the GPU. The rigidity of the solver design leaves little margin for modifications. Code complexity is high and the period of vocational adjustment for modifications can be up to several months. The level of complexity increases by integrating many–core hardware support, mainly because CPU and many–core accelerator do not share the same memory space.

The remaining paper is organized as follows: Sec. 2 gives a brief introduction to directive–based hardware accelerator frameworks. Sec. 3 is an outline of the case study. Sec. 4 and Sec. 5 are dedicated to results and remarks on a further acceleration. The paper finishes with a conclusion in Sec. 6.

2 Directive–based Hardware Accelerator Frameworks

Directive–based hardware accelerator frameworks allow many–core programming by adding meta–information in form of directives to CPU code blocks. The frameworks generate accelerator code based on this information. Fig. 1 shows the basic concept and indicates that large parts of accelerator programming are transparent to the programmer. An API and data–directives allow to control the accelerator and data transfers between system memory and accelerator. Membarth et al. performed an exhaustive evaluation of such frameworks for a small scale code in image registration [4] obtaining good results.

As of today, two major frameworks are established. OpenACC is an open standard by a consortium of numerous vendors [5]. It can be considered a consequence of PGI Accelerator by the Portland Group. The PGI implementation recently added support for Intel Xeon Phi processors. OpenHMPP is mainly directed by CAPS Enterprise and one implementation is HMPP Workbench [6]. OpenACC is more intuitive and strongly relies on a sophisticated code generator but lacks of flexibility compared to HMPP. The frameworks are accompanied by a porting methodology for existing codes. Due to restrictions given by hardware and frameworks, this methodology is applicable to a rather narrow class of code without major changes in software architecture and coding style.

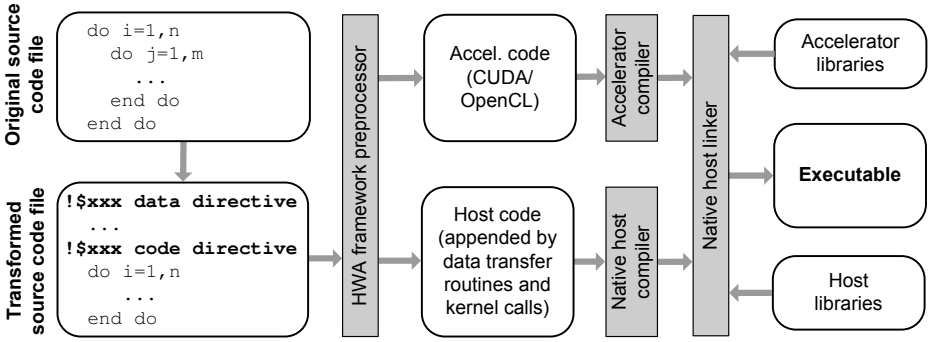

Fig. 1. Basic operation mode of directive–based frameworks.

There are efforts to eventually adopt OpenACC in the OpenMP standard. A recent technical report specifies accelerator directives proposed for a future OpenMP standard [7].

3 The Case Study

Subject to our case study is a many–core acceleration of an industrial MPI–based CFD solver for structured meshes and various CFD models. Restrictions as presented in Sec. 1 make directive–based frameworks the best option for an acceleration for now. The time frame for the case study was 10 months. It was split in 3 months for the evaluation of the best suited framework and for code analysis followed by 7 months for porting and testing.

3.1 Code Foundation

The solver has reached an advanced state of maturity and is in development for 20+ years. It is based on finite volume schemes, multilevel methods and distributed memory parallelization. It is implemented in the Fortran 77 language and the full code size can be estimated to be 400 K lines. Each CFD model has a separate, possibly redundant implementation.

Large initialization routines at program launch read in the configuration such as CFD model, geometry and load distribution amongst multiple processes for the simulation from the hard disk. A Fortran 77 work array gets allocated based on this configuration. It contains all data required for the particular CFD model. It further contains control flags for large mediator routines and space for temporary data, which is managed on a stack in this array. This results in spaghetti code. However, it can be considered representative for a number of industrial codes.

The work array is argument to all mediator routines in the solver and data within the array may be accessed by means of hash functions. The mediator

routines control the program flow, e.g., make decisions about the right compute routines for a CFD model and allocate/deallocated temporary data on up to seven levels. Multi grid cycles within global iteration and time marching loops are performed until convergence after initialization. The dependent variables are advanced one time step per iteration in which fluxes and residuals are computed and boundary elements are communicated to other processes in parallel execution.

3.2 Approach and Work Done

An evaluation of HMPP Workbench and PGI Accelerator (similar to OpenACC) based on a coupled 7-point-stencil computation, as it may arise in finite volume schemes, showed that HMPP Workbench gives better performance results (Tab. 2). The compelling argument for excluding PGI Accelerator is the lack of applicability to multi grid as data transfers cannot be avoided between levels.[1] The solver prove to be sensitive to the many–core integration, resulting in side effects hard to resolve on high level routines. Thus, a minimal invasive bottom–up approach was chosen. We started instrumenting flux routines and established data transfers before and after they are called. Constant data is transferred only once. The flux computations are implemented in a face–centered scheme. In a first step we compute the fluxes of the faces in two/three iterations. Each iteration computes all faces of one spatial direction with a mapping of one thread per face. In a second step we compute the residuals by summing up the fluxes. The porting approach for these low–level routines is similar to the porting methodology indicated by framework vendors (e.g. [6]).

All code related to the accelerator was then offloaded to an external Fortran 90 module. This allows to use pointers for data management and establishes a separation from CPU code.

3.3 Confinements During the Porting Process

During the porting process we observed major difficulties. They can be characterized in three categories: Software design, framework–related issues and limitations given by the hardware. Crucial points are discussed below. Issues marked with † are fundamental and related to technological limitations. Issues marked with ⋆ are related to software design and may be resolved by manpower.

⋆*The Solver Architecture* does not implement a separation between logic/program flow, data and algorithms. This can be resolved at the end of the call tree, but not a priori for other routines. Hence, the percentage of code that can be accelerated is limited. A reorganization is expensive. Adding additional program logic to resolve this is error–prone and hard to maintain.

[1] The same statement holds for OpenACC 1.0. A recent proposal for OpenACC 2.0 resolves this issue.

†*Hardware Accelerator Frameworks* like HMPP Workbench are a rather new piece of technology. They have reached some maturity but cannot be compared to well established compiler suites in terms of stability and functionality. This also holds for debugging. In our case, all accelerator related code is part of a Fortran 90 module. A modification of one code line requires a recompilation of the full module and results in large compilation times. Certain e.g. GPU specific features are not supported. This has an impact on the performance achievable.

†*Bit–compatibility* as binary equivalence of arithmetic results to previous versions of the solver on the same hardware architecture is of big importance for the vendor and demanded by some customers. For internal testing purpose it is considered to be valuable. By invoking many–core hardware it cannot be preserved. Even minor code modifications, which are mandatory for a good many–core performance, are likely to cause different results at bit–level.

†*Compute Performance* of one code targeting two different hardware architectures is a tricky matter. A directive–based framework creates accelerator code by means of CPU code. A compromise between CPU and many–core performance is evident. Code redundancies for CPU and many–core hardware for large parts of the solver reduce the application of directive–based frameworks to absurdity. In general, CFD algorithms for CPU execution are well studied. This is not true for for many–core hardware, except for standard algorithms.

⋆*Data Management and Data Transfers* to accelerator memory have to be performed explicitly and depend on the CFD model launched. An increasing number of routines accelerated requires complicated if/else constructs to decide when and which data has to be allocated/transferred.

Consequences are that the issues above may not be resolved by so called workarounds. Doing so is errorprone, reduces the reliability and maintainability of the solver and would result in fragile code.

4 Results

4.1 Framework Evaluation and Compute Routines

Tab. 1 shows the hardware used for benchmarking. We always use 1 core for CPU execution. Tab. 2 shows a performance evaluation of hardware accelerator frameworks for a coupled 7–point stencil arising from a CFD flux computation on system (A). After familiarization with the frameworks, approximately one man week was invested for each implementation. The better performance of HMPP compared to CUDA for large grids can be explained by this time frame. In CUDA the memory wall can be shifted for small grids by texture caches. HMPP gives better results than PGI Accelerator. The table shows that memory transfers are the main bottleneck reducing the best speedup from 21.7 to 1.6.

	(A)	(B)
CPU	Intel Core i7 2600K	Intel Xeon X5650
# CPUs	1	2
Cores used	1	1
Clock [GHz]	3.40	2.67
Memory [GB]	16	96
GPU	GTX 580	Tesla C2070
# GPUs	1	4
Compiler	gcc 4.5	gcc 4.1
HMPP Workb.	3.1	3.1
PGI Accel.	12.04	12.04

Table 1. Description of the hardware platforms used.

Grid points	CPU	PGI	HMPP	CUDA	PGI speedup	HMPP speedup	CUDA speedup
	[ms]						
Excluding data transfers							
61 712	6.7	0.7	0.7	0.2	9.6	9.6	**27.9**
474 525	29.2	2.4	1.8	1.5	6.2	16.2	**19.5**
1 868 907	115.2	6.7	5.3	6.1	17.2	**21.7**	18.9
Including data transfers							
61 712	6.7	38.8	4.6	n/a	0.2	**1.5**	n/a
474 525	29.2	53.6	22.3	n/a	0.5	**1.3**	n/a
1 868 907	115.2	96.4	72.9	n/a	1.2	**1.6**	n/a

Table 2. CPU and GPU timings and speedups for a coupled 7–point stencil (single precision).

Tab. 3 shows timings for routines accelerated. An acceleration pays off only if data transfers can be reduced between calls for several routines. This is not true for routines with high arithmetic density. We could limit the CPU performance loss to an acceptable level. The CPU run time of Flux 2 could be reduced significantly.

Routine	CPU orig. [ms]	CPU mod. [ms]	CPU loss/gain [%]	GPU excl. transf. [ms]	GPU incl. transf. [ms]	Speedup CPU orig./GPU excl. transf.	Speedup CPU orig./GPU incl. transf.
Flux 1	115.2	118.0	-2.4	5.3	72.9	21.74	1.58
Flux 2[1]	133.0	49.9	+265.3	2.4	n/a	20.79	n/a
Flux 3	503.5	515.0	-2.3	13.7	43.4	36.75	11.60
Flux 4	1019.5	1034.5	-1.45	15.9	46.2	64.12	22.06
Flux 6[2]	2149.5	2195.6	-2.1	17.0	47.5	126.44	45.25
Flux 7	n/a	817.3	n/a	27.1	76.5	30.16	10.46
Precon. 1	n/a	273	n/a	10.0	51.0	27.30	5.35
Tridiag.	27.0	31.1	-15.1	2.3	24.0	11.74	1.13

Table 3. CPU and GPU timings and speedups for the accelerated routines excluding and including data transfers with a block size of 149x113x113.

4.2 Test Case

Tab. 4 shows timings for a CFD computation for an annular tube based on a steady–state laminar Navier–Stokes model. The mesh consists of seven blocks, each having three levels of discretization (a total of 500 MB memory). The finest discretization is computed on the GPU, the two remaining discretizations

[1] Here we calculate the GPU speedup based on the modified CPU implementation because we could accelerate the CPU code by a factor of approx. 2.5.

[2] Dense arithmetics with few memory accesses and few data transfers is performed in this routine. Still, there may be room for improvements of the CPU code.

are computed on the CPU because coarse discretizations penalize GPU performance strongly. One can observe that the routines instrumented take approx.

	CPU orig. [s]	CPU mod. [s]	CPU Perf. Loss/Gain [%]	GPU [s]	Speedup CPU orig./GPU
Total run time	1303	955	+36	784	1.66
Incl. data transfer					
Flux 1	91.91	90.71	+1.32	74.76	1.23
Flux 3	134.89	132.80	+1.58	21.78	6.19
Precond. 1	415.08	140.91	+294.57	67.98	6.11
Excl. data transfer					
Flux 1	91.91	90.71	+1.32	17.97	5.11
Flux 3	134.89	132.80	+1.58	9.45	14.27
Precond. 1	415.08	140.91	+294.57	11.43	36.31

Table 4. CPU and GPU timings for a sequential computation of an annular turbine (laminar Navier-Stokes).

65 % (35 %) of the total run time of the solver for the original (modified) CPU code. The maximal achievable speedup is less than 3 (1.5) compared to the original (modified) CPU code. We could achieve an overall speedup of 1.66 for the computation compared to the original CPU code. The modified CPU code runs faster than the original CPU code on (A).

A parallel execution of the test case on test system (B) with 1 master process and three compute processes (each having 1 GPU attached) gave a total speedup of 1.10. The MPI load–balancing strategy used for CPU execution did not prove to be suitable for GPUs, as their performance has a highly nonlinear relation to the size of domains processed.

5 Long Term Approach

The results show, that the acceleration of hot spots only results in a modest performance gain. Data transfers between CPU and accelerator are the main bottleneck. They can be reduced if all compute related routines are accelerated and share data without passing it via CPU memory first. For this, these routines must be prepared for many–core execution and reimplemented for a latter application of HMPP directives. Then it suffices to only transfer boundary elements within a multi grid cycle for MPI–based parallelism. We now outline an approach for a full acceleration. It may be applicable for a many–core acceleration of similar Fortran codes that use a work array for dynamic data management.

1. Implement automated accelerator data allocation and management based on the compute configuration instead of a manual data allocation in mediator routines. We consider this to be essential for the code integrity of the solver.
2. Perform a full many–core acceleration of a simple test case. I.e.:

  - For each compute related routine invoked in the test case:
    - Separate computations from logic and logging/error handling, i.e., define code parts for accelerator execution.
    - Instrument the routine by means of HMPP Workbench.
    - If the routine shares data with other routines already instrumented, omit redundant CPU–accelerator data transfers.
3. Enable asynchronous execution for independent compute routines to fully occupy the accelerator.
4. Refine the support for MPI–based parallelism. (CPU–accelerator transfers of boundary elements).
5. The repeated porting of code invoked for other test cases.

One consequence is the reimplementation of the solver in large parts for many–core execution. Some of the fundamental aspects discussed in Sec. 3.3 still can not be tackled. We estimate labor costs of approximately 24 man months for an acceleration of a rather simple CFD model within the solver and a speedup for this model of 20 at best.

6 Conclusion

We performed a case study for a many–core acceleration of a large–scale numerical CFD code by means of directive–based frameworks. Hot spots could be accelerated significantly, whereas the overall performance gain is modest.

The frameworks are convenient, as they reduce implementation time for accelerator code dramatically. But the major challenge in software engineering is not necessarily implementation work, but also algorithm and software design. This is particularly true for many–core hardware, as it is not fully compatible with well–established software engineering techniques: Constraints due to solver architecture, complexity and technological limitations prevent from a further acceleration of the solver without intricate work. In our case, many–core hardware and large code complexity can be hardly combined.

The usability of the frameworks is high for new or small scale codes that are primarily dedicated to many–core hardware. In such cases the limitations of hardware and frameworks can be met, without having to consider aspects related to CPU execution or existing code designs. For other cases, open questions have to be answered by hardware and framework vendors.

Acknowledgments. This work is supported by the CleanSky Joint Undertaking trough grant JTI–CS-2010–1–GRA–02–008 within the Seventh Framework Programme of the European Union.

References

1. nVidia Corporation: CUDA CUBLAS Library. (August 2010)

2. Micikevicius, P.: 3D finite difference computation on GPUs using CUDA. In Kaeli, D.R., Leeser, M., eds.: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. ACM International Conference Proceeding Series, ACM (2009) 79–84
3. Emans, M., Liebmann, M.: Velocitypressure coupling on gpus. Computing **94** (2012) 1–21
4. Membarth, R., Hannig, F., Teich, J., Korner, M., Eckert, W.: Frameworks for gpu accelerators: A comprehensive evaluation using 2d/3d image registration. In: Proceedings of the 2011 IEEE 9th Symposium on Application Specific Processors. SASP '11, Washington, DC, USA, IEEE Computer Society (2011) 78–81
5. Cray Inc., CAPS Enterprise, NVidia, Portland Group: OpenACC 1.0 Specification. (2011)
6. CAPS Enterprise: HMPP 3.2 Workbench Directives. (2012)
7. OpenMP Consortium: OpenMP technical report 1 on directives for attached accelerators. Technical report (November 2012)