Performance Portability of OpenMP

Daniel Malik

Technical University of Munich

July 5, 2022

Abstract

In recent years, the focus in HPC has changed towards accelerator devices such as GPUs. Because of this change, the demand and importance of performance portability is steadily increasing. Developers want to use their code on a variety of different devices without having to make many specific changes to it. Additionally, the performance and efficiency of the code should remain similar after porting. In this paper we are taking a closer look at OpenMP and its base functionalities before analyzing its performance and portability and comparing it to different other options.

1. Introduction

The OpenMP API was originally created in the 1990s by a group of vendors. Its main purpose was to standardize notations for specifying how tasks should be mapped to individual processors. The first version could only be used with Fortran while support for C and C++ was added later. Since then, OpenMP developed to a modern, easy to use API that can be utilized to specify high-level parallelism on shared memory multiprocessors. Apart from a set of compiler directives it also contains environmental variables and library routines to do so. To date OpenMP is still supported by major CPU and GPU vendors as well as many popular compilers.[1]
In the second section of the paper, we first present how to implement OpenMP into an existing sequential program. For Subsection 2.2 we explain the basic functionalities of the API before presenting multiple supported types of parallelism in Subsection 2.3. The third section compares sequential code with its parallelized counterpart to describes and analyze the performance of OpenMP. Section 4 focuses on the portability of OpenMP source code while specializing on GPU offloading in Subsection 4.1. The performance portability is evaluated in Subsection 4.2 and OpenMP's role on modern supercomputer architecture is presented in Subsection 4.3. In the fifth section we compare OpenMP to other options that can be used to specify parallelism.

2. Usage of OpenMP

2.1. Implementation

One of the key reasons for OpenMP's popularity is its simplicity. We first want to look at a base implementation before introducing additional features that can be added to further improve parallelization and performance.
OpenMP is built into the compiler so there are no additional libraries that need to be installed. The API can be utilized by simply adding the header in C and C++ or using the module for Fortran. Programmers can then insert high-level directives into the sources code as comments in Fortran or pragmas in C/C++. The low-level details are hidden, making the code more compact and easier to read. Finally, to run the code with OpenMP enabled it must first be compiled with the appropriate flag. There is a wide range of compilers that support OpenMP most notably GCC or Clang.[2] Listing 1 shows a simple parallelized "Hello, World!" program with OpenMP. How often "Hello, World!" is written to standard out depends on the number of cores of the CPU.

.

Listing 1: *Parallel "Hello, World!" with OpenMP*

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
   #pragma omp parallel
    {
    printf("Hello, World!\n");
    }
  return 0;
}
```

2.2. Functionality

Apart from the creation of threads, OpenMP offers more parallelization functionalities. In this subsection we want to introduce some of these main features for parallelizing more complex programs. The OpenMP directives allow programmers to:

- differentiate between parallel and serial regions
- define how loops should be parallelized
- state if variables are supposed to be private or shared
- specify how the work is scheduled between threads

In regards of memory management OpenMP offers classifiers for private and shared variables. When using private ones, a copy of them is created for each process while shared variables have one instance for all processes. Variables created within parallel regions are private by default while the ones created in sequential regions are shared by default.[3]
When using shared variables there is always the risk of conflicting accesses. OpenMP uses event synchronization and mutual exclusion as well as other synchronization constructs as prevention. Mutual exclusion gives a thread exclusive access to a variable while event synchronization can be established by so called "barriers" in the code at which threads have to wait until all other threads have arrived. To avoid simultaneous writing to memory the atomic construct can be used to force any memory update in the next instruction to happen atomically.[3] For our example in Listing 1 the parallel threads writing to standard out can possibly create a race condition leading to unintended outputs. To fix this issue we could add a Mutex for standard out.
Additionally, when defining parallel parts of the serial code the user can incrementally parallelize with OpenMP. The high-level directives can be inserted only into some areas of the code while leaving others sequentially. If the code compiles and tests on it were successful another portion can be parallelized. This process can be repeated until the required speedup is achieved.[2]

2.3. Types of parallelism

OpenMP supports different types of parallelism. Generally, these can be grouped into data and task parallelism. Data parallelism uses the same instructions on different data simultaneously while task parallelism splits the code into tasks that are then computed in different threads. A well-known type of data parallelism is loop-level parallelism. A typical example is a loop iterating over an array while modifying the entries independently. Because of this independency the order of the iterations is irrelevant and therefore they can be split on several threads and executed simultaneously. The same can be done for independent nested loops such as an iteration over a multidimensional array. OpenMP offers a clause to collapse the two loops into one which can improve the run-time performance by removing loop overhead. Another supported type is nested parallelism. If a region of the code is already executed in parallel by a team of threads and one of them encounters a parallel construct the thread can create a new team becoming the master of that team. An example where nested parallelism is often used are recursive algorithms.[1]

3. Performance

After getting a basic understanding of OpenMP's core functionalities we now want to analyze its performance compared to a serial program. The improve in performance is highly dependent on the type of algorithm that is to be parallelized as well as the architecture the code is executed on. Therefore, the following example feature two different algorithms that were executed on hardware that enables strong parallelization.
The performance-benchmark [4] that was executed on two 24-core Intel Xeon Platinum 8160 CPUs analyzed the Speedup using task parallelization on two mathematical procedures. The performance of OpenMP's loop construct was compared to task-based implementations both with and without dependences. The dependences allow specified tasks to start operating immediately after the prior tasks they depend on have finished. This is an upgrade compared to the common task-based implementation which need to use the barrier construct introduced earlier. Figure 1 shows the performance in relation to a serial implementation highlighting the advantage that dependences can provide as well as displaying how well OpenMP can speed up the computation of these highly parallelizable algorithms.

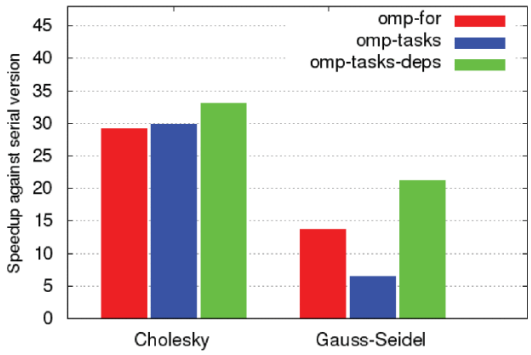

Figure 1: *Performance comparison: Loops, Tasks and Dependences to serial; 2x 24-core Intel Xeon Platinum 8160 CPUs [4]*

4. Portability

To compute algorithms that allow strong parallelization even more efficiently, programmers must resort to hardware other than CPUs, like accelerator devices. Even though OpenMP was initially designed for parallelization on CPUs, the support for offloading Code to GPUs was added with OpenMP 4.0 in 2013.[5] In the following three subsections we want to introduce GPU offloading with OpenMP and compare its performance to its non-ported version on the CPU as well as discuss OpenMP's role on supercomputers.

4.1. GPU offloading

While CPUs are optimized for serialized tasks GPUs have many cores for enhanced parallel throughput computations. While design can vary greatly depending on the hardware manufacturer and version of the GPU, a typical one consists of multiple streaming multiprocessors each containing many cores collaborating via shared memory. The cores themselves have thread blocks consisting of multiple threads. Usually, threads of the same block can only perform the same task simultaneously so they can exclusively be used for data parallelism while task parallelism can be done between the cores or streaming multiprocessors.[6]
Porting code to a GPU can be done with the target construct of OpenMP. This directive brackets an executable region to be run on the offloading device. Within the target region the data access is specified. This is necessary because GPUs have their own memory to which the data must be copied before calculation. In OpenMP this can be done by using the map clauses with the following syntax:

$$map(map-type : list)$$

where list can be a pointer to the data section or an array. For the latter, an array section can be specified. The map-type can be one of the

Table 1: *Compilerflags [8]*

Flags	Clang/Cray/AMD	GCC/GFortran
OpenMP flag	-fopenmp	-fopenmp
Offload flag	-fopenmp-targets=<target>	-foffload=<target>
Target NVIDIA	nvptx64-nvidia-cuda	nvptx-none
Target AMD	amdgcn-amd-amdhsa	amdgcn-amdhsa
GPU Architecture	-Xopenmp-target -march=<arch>	-foffload="-march=<arch>

following:

- to – copy data to the device at the start of the target region
- tofrom – copy data to the device at the start of the target region and from the device at the end of it
- from – copy data from the device at the end of the target region
- alloc – created space for input on the device at the start of the target region
- delete – delete allocated data on the device

The default map-type is tofrom.[6]
The compiler additionally must be instructed to generate GPU-specific code and the target GPU architecture must be specified. To do so the right compiler flags must be set on compilation. OpenMP is supported by major GPU vendors like Nvidia or AMD. For those two Table 1 taken from an article [7] shows the appropriate compilation flags for different compilers.

4.2. Performance portability

Performance portability is key when offloading work to the GPU. If the code does not run efficiently on the target architecture, the main reason for offloading is gone.
In an article [8] from 2019 the performance portability was evaluated using 4 NVIDIA Tesla V100 GPUs with 16 GB memory each. The GPUs competed against an IBM Power9 CPU. For the first test the evaluation of the following equation with 3 single-dimensional arrays is used as an example for a memory-intensive benchmark:

$$A[i] = B[i] + constant * C[i]$$

Due to the high amount of memory transfer between the CPU and GPUs overshadowing the actual compute part the CPU clearly outperformed the GPUs regardless of the selected array size. This benchmark emphasizes the weakness of GPU offloading.
As a second benchmark a simple two-dimensional matrix multiplication which is both compute and memory intensive was executed on the same hardware. While the CPU was still faster for smaller matrices the GPUs caught up at a size of about 1300 x 1300. Due to the large amount of possible parallelization, the required time of the GPUs remained constant up to very large matrices while the execution time on the CPU increased by $O(N^3)$. For a 3000 x 3000 matrix a speedup of 6 was archived.
Another benchmark done in a Case Study for Performance Portability [6] used OpenMP 4.5 on a P100 GPU to implement a General Plasmon Pole (GPP) kernel for a comparison between an atomic implementation and one with reduction clauses from OpenMP. The GPP kernel featured 3 nested for-loops where the two most outer loops of the kernel were collapsed and parallelized over the thread blocks of the GPU. The iterations of inner loop were distributed to the individual threads. With the kernels main computational bottlenecks being dense linear algebra, large reductions, and fast Fourier transformations a speedup of 3 was achieved compared to a serial implementation.

4.3. Modern supercomputers

For even greater scaling, modern supercomputer architecture is the limit. A typical super-

computer has multiple compute-clusters containing multiple nodes consisting of several sockets, each of which has a CPU with multiple processor cores. The clusters and nodes are coupled via high-speed interconnects while the CPUs within a node use shared memory. This type of architecture has multiple levels of parallelism. When using OpenMP alone on one of these architectures the problem arises that there is only distributed memory between the nodes and clusters. There are approaches to use OpenMP with these types of systems as well but the partitioning and placing of data onto the distributed memories must be done separately for example with MPI. This forms a so-called hybrid model where OpenMP is used for parallelizing within the nodes and MPI for communication between them. Another option is a in 2006 developed compiler add-on by Intel called "Cluster OpenMP". The add-on enables the user to create OpenMP flush points automatically or manually at which memory pages are kept coherent between the nodes by a protocol.[9]
To evaluate the scalability of OpenMP for supercomputers, a paper [10] evaluated the performance of a disjunctive normal form (DNF) algorithm to compute the power set on the Skylake compute node of the Stampede2 supercomputer. The authors showed that the speed-up of the DNF algorithm with OpenMP scaled linear with the number of Cores. This example emphasizes the scalability of OpenMP up to large numbers of cores.

5. Comparison

The OpenMP API is by no means the only tool available for realizing parallelization and offloading Code to accelerator devices. To put OpenMP in a broader context we will now compare it with two other popular options.

5.1. OpenACC

OpenMP and OpenACC are very similar in many regards. Both are directive APIs for parallel programming, support similar types of parallelism and the same programming languages. They share their simplicity, are supported by most hardware and compilers, and perform similar in most benchmarks. A performance analysis [11] of CUDA, OpenACC and OpenMP on a TESLA V100 GPU showed similar results for both OpenACC and OpenMP with a slight lead for OpenACC.
Even though OpenMP and OpenACC have much in common, the two APIs were initially built for different purposes. OpenMP was designed to parallelize Code on CPUs while OpenACC was created for accelerator devices. Because of this OpenMP gives the compiler and optimizer less flexibility, leaving parallelization and scheduling responsibilities to the programmer. A typical decision that the compiler can still make is to determine how many threads to create if the programmer does not specify otherwise. OpenACC on the other hand allows the compiler to choose between common parallelization options like SIMD or threads based on the underlying hardware. Nevertheless, OpenMP's offloading features have improved in recent years, offering an available alternative to OpenACC.[12]

5.2. CUDA

Compared to OpenMP, CUDA is a low-level programming model. The prior named performance analysis [11] shows that the performance of both CUDA and OpenMP is similar for simple parallel test cases. For more complex code on the other hand a significant gap was observed. A test case on sum reduction memory access patterns showed that OpenMP was up to 80% slower than CUDA. Especially on large data inputs OpenMP and OpenACC struggled to keep up with CUDA. The authors of the performance analysis believe that compiler optimization is the key of closing that gap.
Because of its low-level programming model CUDA is far more complex to use. It requires prior knowledge and a deeper understanding. To port an existing C program, the programmer needs to make changes directly into the

code, for example to function-calls for memory allocation, resulting in much higher porting effort. Another big drawback is the fact that CUDA is limited exclusively to NVIDIA hardware. The choice between CUDA and OpenMP therefore depends mostly on the architecture as well as the kind of application.[12]

6. Conclusion

In this paper we summarized the base functionality of OpenMP before focusing onto portability to modern supercomputers and especially GPUs. The performance of OpenMP optimized code was compared to sequential code showing huge potential for speedup. Finally, a comparison was drawn with OpenACC and CUDA highlighting that despite weaker performances OpenMP can score its simplicity and general support on many types of hardware. OpenMP has not lost its relevance in the parallelization of programs to this day. Despite its age, the OpenMP ARB ensures that OpenMP is supported on the latest hardware making it a safe option for many years to come.

References

[1] M. van Waveren, M. Klemm, M. Wong, J. Hoeflinger, A. Fritsch, K. Mattson, R. Friedman (2022). "OpenMP FAQ". openmp.org. https://www.openmp.org/about/openmp-faq/#Document (accessed June. 27, 2022).

[2] B. Chapman, G. Jost, R. van der Pas (2008). "Using OpenMP". pdfs.semanticscholar.org. https://pdfs.semanticscholar.org/932d/5abe3d49f3c49d77c6e60ddbd0e3dfcae8dd.pdf (accessed June. 27, 2022).

[3] OpenMP Architecture Review Board (2021). "OpenMP Application Program Interface". openmp.org. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (accessed May. 24, 2022).

[4] B. de Supinski, T. Scogland, A. Duran, M. Klemm, S. Bellido, S. Olivier, C. Terboven, T. Mattson (2018). "The Ongoing Evolution of OpenMP". ieeexplore.ieee.org. https://ieeexplore.ieee.org/abstract/document/8434208 (accessed June. 26, 2022).

[5] S. Bak, C. Bertoni, S. Boehm, R. Budiardja, B. Chapman, J. Doerfert, M. Eisenbach, H. Finkel, O. Hernandez, J. Huber et al. (2022). "OpenMP application experiences: Porting to accelerated nodes". sciencedirect.com. https://www.sciencedirect.com/science/article/pii/S0167819121001009 (accessed June. 13, 2022).

[6] R. Gayatri, C. Yang, T. Kurth, J. Deslippe. "A Case Study for Performance Portability using OpenMP 4.5". sc18.supercomputing.org. https://sc18.supercomputing.org/proceedings/workshops/workshop_files/ws_waccpd109s2-file1.pdf (accessed May. 31, 2022).

[7] F. Robertsén, O. Louant, G. Markomanolis (2021). "Offloading code with compiler directives". ilumi-supercomputer.eu. https://www.lumi-supercomputer.eu/offloading-code-with-compiler-directives/ (accessed June. 17, 2022).

[8] A. Nitsure, H. Shrivastava, P. Dsouza (2019). "GPU programming made easy with OpenMP on IBM POWER – part 1". developer.ibm.com. https://developer.ibm.com/articles/gpu-programming-with-openmp/#ref1 (accessed June. 12, 2022).

[9] G. Hager, G. Jost, R. Rabenseifner (2009). "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes". www3.nd.edu. https://www3.nd.edu/~zxu2/acms60212-40212-S12/Cray09-hybrid-MPI-OpenMP.pdf (accessed May. 31, 2022).

[10] R. Goodwin (2021). "Linearizing Computing the Power Set with OpenMP". ieeexplore.ieee.org. https://ieeexplore.ieee.org/document/9460698 (accessed June. 21, 2022).

[11] M. Khalilov and A. Timoveev (2021). "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU". iopscience.iop.org. https://iopscience.iop.org/article/10.1088/1742-6596/1740/1/012056 (accessed June. 17, 2022).

[12] R. Usha, P. Pandey, N. Mangala (2020). "A Comprehensive Comparison and Analysis of OpenACC and OpenMP 4.5 for NVIDIA GPUs". ieeexplore.ieee.org. https://ieeexplore.ieee.org/document/9286203 (accessed June. 19, 2022).