Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU

To cite this article: Mikhail Khalilov and Alexey Timoveev 2021 *J. Phys.: Conf. Ser.* **1740** 012056

Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU

Mikhail Khalilov[1,2], Alexey Timoveev'2,1

[1] HSE University, Myasnitskaya 20, Moscow 101000, Russian
[2] Joint Institute for High Temperatures of the Russian Academy of Sciences, Izhorskaya 20 bld.2, Moscow 125412, Russia

E-mail: `mkhalilov@hse.ru, timofeev@jiht.ru`

Abstract. Graphics processors are widely utilized in modern supercomputers as accelerators. Ability to perform efficient parallelization and low-level allow scientists to greatly boost performance of their codes. Modern Nvidia GPUs feature low-level approaches, such as CUDA, along with high-level approaches: OpenACC and OpenMP. While the low-level approach aims to explore all possible abilities of SIMT GPU architecture by writing low-level C/C++ code, it takes significant effort from programmer. OpenACC and OpenMP programming models are opposite to CUDA. Using these models the programmer only have to identify the blocks of code to be parallelized using pragmas. We compare the performance of CUDA, OpenMP and OpenACC on state-of-the-art Nvidia Tesla V100 GPU in various typical scenarios that arise in scientific programming, such as matrix multiplication, regular memory access patterns and evaluate performance of physical simulation codes implemented using these programming models. Moreover, we study the performance matrix multiplication implemented in vendor-optimized BLAS libraries for Nvidia Tesla V100 GPU and modern Intel Xeon processor.

1. Introduction

Graphics processing unit (GPU) along with central processing unit (CPU) plays a major role in boost of computing nodes performance in 6 out of 10 supercomputers from November 2019 TOP500 rating. GPUs allow scientific applications to greatly increase performance using fine-grained parallelism. At the same time, writing optimized and scalable code for GPU is a very difficult task for a programmer. Moreover, there are a lot of different paradigms and application programming interfaces (API) for GPU programming. So, the performance study of these paradigms may significantly help the programmer to make the best choice. OpenMP [1] and OpenACC [2] frameworks make attempt to simplify and abstract parallelization of code using compiler pragmas. On the other hand, CUDA still remains to be the most popular API for writing low-level parallel code for GPUs. Almost one third of TOP500 supercomputers use graphics cards (GPUs) as accelerators. Modern GPUs support a wide range of parallel programming models. Some of these models, like CUDA and OpenCL are designed for low-level optimization in C/C++ language. While such low-level approach makes possible it to use all capabilities of SIMT architecture, it still remains very difficult.

The piece of code that will be executed in parallel on the GPU hardware CUDA cores is called the kernel. CUDA allows the programmer to directly manipulate hardware and software GPU resources within the kernel such as threads, blocks of threads, grids of blocks, memory and cache allocation, thread registers, asynchronous thread execution, etc. On the other hand, writing fast, low-level code to take into account all these nuances of the GPU architecture is a very time-consuming task for programmer.

Contrary to the low-level approach to parallelization, OpenACC [2] and OpenMP [1] represent a declarative model of parallel programming using compiler pragmas. Using OpenACC and OpenMP, the task of generating parallel code for the GPU lies with compilers, which are not always able to cope with this task effectively. For this reason, a programmer will often be forced to write code that conceptually repeats CUDA or OpenCL code when creating a program to get maximum performance on OpenACC and OpenMP.

In this work, the performance study of various parallel programming models is performed by running various GPU benchmarks. Naive and manually optimized matrix multiplication algorithms have been developed. The reason for choosing this matrix operation is to study the performance of well-parallelized algorithms used in real-world applications.

Performance of Nvidia's core linear algebra routines (BLAS) libraries (cuBLAS) on GPUs and Intel (Math Kernel Library) for Intel CPUs is also analyzed in addition to testing our custom matrix multiplication implementations. BLAS performance can set an upper bound on the performance of GPU optimized code, since cuBLAS and MKL are optimized by hardware manufacturers for use in commercial applications.

Performance study of several real-world applications parallelized in OpenACC and CUDA is also presented.

2. Related work

In many works [13], [14], attempts have been made to comparatively analyze various parallel programming models for GPUs and other accelerators.

Authors of [15] try to compare CUDA, OpenMP and OpenACC implementing parallel computation of aerial target reflection of background infrared radiation using these models. Direct comparison of OpenMP and OpenACC on standard parallel programming patterns (map, stencil, reduction, fork-join etc.) is done in [16]. Papers [7, 8] are devoted to the design of cost-effective GPU-based clusters for molecular dynamics simulation.

At the same time, the authors of this work were not able to find works where CUDA, OpenMP and OpenACC are directly compared under the same conditions.

3. Methods and Software Implementation

3.1. Matrix - matrix multiplication as a benchmark

In this paper, the parallelization efficiency was estimated by measuring the performance of the operation of matrix-matrix multiplication:

$$C = \alpha * A * B + \beta * C$$

— called `sgemm` function in specification of the BLAS [17] .

In real tests, for convenience, the numbers of columns and rows of A and B are equal. The elements of A and B is initialized with pseudo-random values ranging from 0.0 to 1.0.

Computational complexity of the `sgemm` function is $\mathcal{O}(2 * m * n * k)$ FP32 operations. m and n are the number of columns of the input matrix A and the number of rows of the matrix B, respectively. k is the number of rows of A and columns B. n_{iter} is the number of calls to `sgemm` to average the calculation time of the result matrix C.

The FP32 metric (the number of single-precision floating-point operations per second (flop/s)) was chosen to measure the GPU performance. Thus, the performance of P_{FP32} is calculated by the formula:

$$P_{FP32}(n, m, k, n_{iter}) = \frac{t_{sum}}{(2 * n * m * k)n_{iter}} \text{flop/s}$$

t_{sum} is total time spent on the n_{iter} iterations. Thus, t_{sum} may include the time taken to manipulate the GPU memory and the time needed for the calculations themselves.

The time to work with the GPU memory routines is the sum of the time to allocate A, B and C arrays, transfer the input data A and B from host memory to the GPU device memory before calculations and C from the GPU device to host memory after calculations and further memory freeing.

GPU device memory buffers for A, B and C was preallocated at the time of the first iteration, so that in subsequent iterations there was no need to allocate it again. In such a scenario, after warm-up iteration t_sum includes the time only the time needed for the calculations.

3.2. BabelStream — GPU memory bandwidth benchmark

The performance of many scientific applications is severely limited by memory bandwidth. For such memory-bound tasks, the memory bandwidth must be large enough to keep time to saturate the computers with data. Otherwise, the CPU / GPU cores will experience ”starvation” and idle waiting for data.

The BabelStream [9], [10] test suite was developed at the University of Bristol to estimate the overhead of various parallel programming models while accessing GPU memory. For each parallel programming technology, including for OpenACC, OpenMP and CUDA, BabelStream, 5 memory-bound tests are implemented:

- Copy: $a[i] = c[i]$
- Multiply: $b[i] = \alpha c[i]$, α - is a scalar constant.
- Add: $c[i] = a[i] + c[i]$
- Triad: $a[i] = b[i] + \alpha c[i]$
- Dot: $sum+ = a[i] * b[i]$,

$a[i]$, $b[i]$, $c[i]$ arrays of float type with double precision.

3.3. Performance evaluation of real applications

Computational algorithms used in real applications are much more complex. For this reason, it’s important to analyze the applicability of various parallelization models to speed up real-world applications. For this, applications were selected for which the parallelization of the computational part was done both using CUDA and with OpenACC:

- Cloverleaf is an application for finding a numerical solution to the Euler equation on a Cartesian 2D/3D mesh [11]. For this application, there are various implementations of parallelization of the computing part, including those using CUDA and OpenACC.
- LULESH - a simplified implementation of the algorithm for finding a numerical solution to the problem of modeling a blast wave in a liquid [12].

3.4. Hardware and software testbed

Performance benchmarking was carried out on the basis of the following hardware:

- **Computing node of National Research University Higher School of Ecomomics Supercomputer:**
  CPU 2x Intel Xeon Gold 6152
  GPU 4x v100 Tesla V100-SXM2 HBM 2 32GB (NVlink), 15.6 TFlops/s
  RAM DDR4-2666 1.5 TB
  OS CentOS 7.

The following software was used during the experiments:

- **OpenACC compiler**: PGI Community Edition 19.10 with flags
`-fast -Minfo=accel -acc -ta=tesla:<compute capability>`
- **OpenMP compiler**: Clang/LLVM 9.0.0 with flags
`-O3 -fopenmp -fopenmp-targets=nvptx64-Nvidia-cuda`
`-Xopenmp-target -march=sm_<compute capability>`
- **CUDA/cuBLAS compiler**: nvcc from CUDA Toolkit 10.0.
- **Intel CPU optimized compiler**: Intel Compiler 2019 (icc) update 3 with flags `-O3 -mkl`

The parameter `<compute capability>` has a value of `70`.

4. Results and Discussion

4.1. Impact of programming model on achievable GPU memory bandwidth.

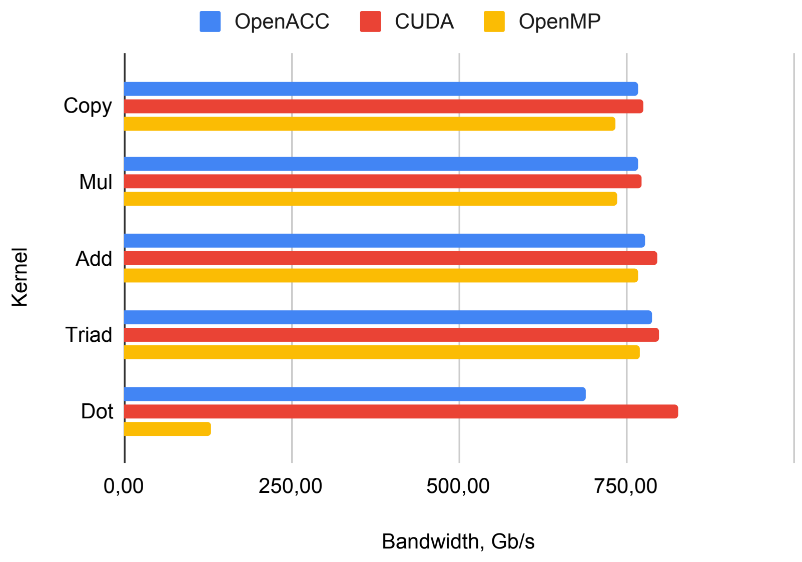

Figure 1. Memory bandwidth achieved on Tesla V100 GPU in BabelStream benchmark using OpenMP, OpenACC, CUDA

On the Copy, Mul, Add, Triad tests, using CUDA it's possible to achieve the maximum bandwidth from 770 Gb/s (85% of the peak) to 800 Gb/s (88% of the peak). OpenACC and OpenMP on the same tests show the same lag behind CUDA with an average of 3-4% and 6-7%, respectively.

On the Dot test ($sum+ = a[i] * b[i]$), CUDA also demonstrates the maximum performance, reaching 91% of the peak memory bandwidth. Dot implementations on OpenMP and OpenACC, using pragmas to optimize the reduction, show the following results:

- OpenMP: `target teams distribute parallel for reduction(+:sum)` - 130 GB / s, 14% of the peak bandwidth;
- OpenACC: `parallel loop reduction(+:sum)` - 690 GB / s, 76% of the peak bandwidth.

These results reflect the tendency that compiler-generated code (PGI for OpenACC, Clang for OpenMP) on less trivial computations is far from the speed of manual CUDA optimization, where one can use the advantages of shared memory. In the future, this hypothesis is confirmed by tests with matrix multiplication.

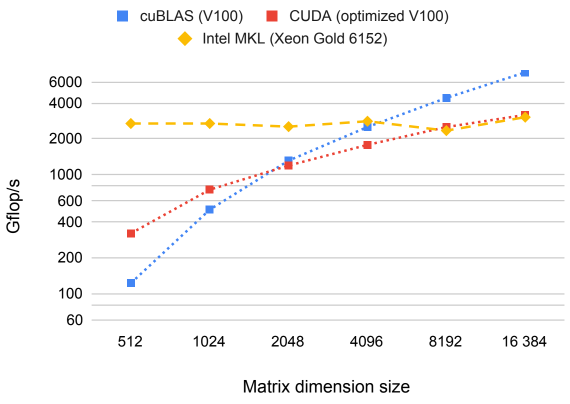

Figure 2. FP32 performance comparison of cuBLAS, Intel MKL and CUDA on Tesla V100

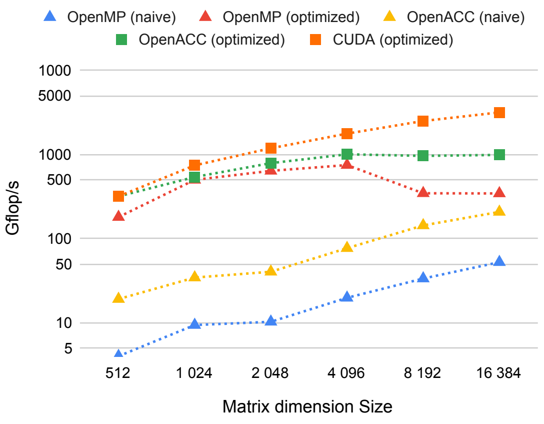

Figure 3. Dependence of FP32 performance on matrix sizes and programming model on Tesla V100

4.2. Performance analysis of optimized BLAS libraries: cuBLAS vs. Intel MKL

Fig. 2 shows the performance results of `sgemm` from Intel MKL, cuBLAS and the hand tuned CUDA implementation.

The number of available threads on a compute node with Intel Xeon Gold 6152 was limited to 22 using the `MKL_NUM_THREADS` environment variable in order to use one physical processor with 22 cores (by default, MKL uses all available processors).

The gap between the Xeon Gold 6152 and the Tesla V100 was 22 times at the input size of 512×512. At the same time, this gap is rapidly narrowing with the increase in input data. The Tesla V100 in conjunction with cuBLAS shows performance superior to Intel Xeon up to 2x times on large data sizes ($> 4096 \times 4096$).

The hand tuned CUDA `sgemm` surpasses cuBLAS by up to 45% on input data sizes of $\leq 2048 \times 2048$ on Tesla V100. The performance of the hand tuned CUDA `sgemm` on large input data becomes comparable with the performance of Intel MKL on Xeon CPUs.

4.3. Comparison of OpenACC, OpenMP, CUDA and cuBLAS performance on matrix multiplication

Figures. 3, show the results of naive and hand tuned `sgemm` implementations on OpenACC, OpenMP and CUDA, as well as the cuBLAS library results on Tesla V100.

The naive OpenMP and OpenACC `sgemm`'s significantly (up to 3x times) lose in performance comparing to naive implementation using CUDA on the Tesla V100 GPU. Optimized versions of OpenMP and OpenACC show results comparable to the naive implementation of naive CUDA `sgemm`. At the same time, as already noted in Section 4.1, with large input data there is a 9x degradation in scalability on OpenMP and 3x on OpenACC compared to CUDA on the Tesla V100.

These optimizations reduced the gap between the tuned OpenMP and OpenACC `sgemm`'s by the 7.5% on the input of size 1024×1024, and by 20% on 2048×2048 input.

Also, with tuned OpenMP `sgemm` there is a performance scaling degradation with an increase of the input size ($> 4096 \times 4096$), until the performance completely stops to grow when comparing 8192×8192 and 16394×16384 input sizes.

A similar effect is observed using OpenACC. This may indicate that the compiler cannot generate offloaded GPU code that optimally scales on the CUDA-cores without further optimization for a specific architecture. For example, using the custom parameters with

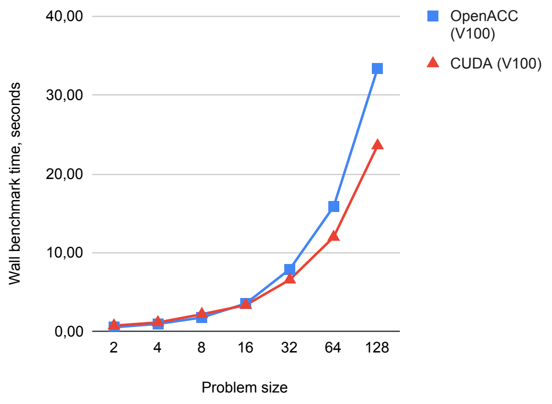

Figure 4. Comparison of OpenACC and CUDA performance on Cloverleaf application

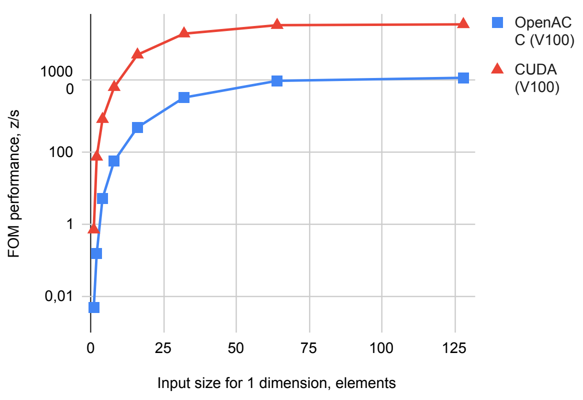

Figure 5. Comparison of OpenACC and CUDA performance on LULESH application

`num_teams(n)` and `num_threads(m)` clauses for OpenMP or `gang(n)` and `vector(m)` for OpenACC.

The tiled matrix multiplication algorithm with usage of shared memory of GPU thread blocks and loop unrolling presented in [6] allows the hand tuned CUDA `sgemm` to win performance up to 1.7x on the Tesla V100.

Using the cuBLAS library, we were able to get as close as possible to the theoretical peak Tesla V100 performance. The performance on the Tesla V100 GPU was 7220 GFlop/s at the maximum input data sizes of 16384x16384 for Tesla V100.

4.4. Comparison of applications performance, parallelized with CUDA and OpenACC

Figure 4 shows the dependence of the total computation time of the Cloverleaf application on the size of the input data.

On small sizes of the computational grid, we observe that calculations using CUDA and OpenACC are performed in a comparable time. As the computing field grows, the CUDA implementation shows better results. So, at the maximum mesh size between CUDA and OpenACC, there is a 40% difference in computation time.

The largest performance difference between OpenACC and CUDA was found when using LULESH code. The input data size for the calculation is specified as the number of cells of a 3-dimensional grid. Figure 5 shows a plot of computation performance (Figure of Merit) versus grid size (along one axis).

Huge performance gap between CUDA and OpenACC implementations observed. The CUDA implementation is up to 100 times faster on a 16 mesh grid. At the maximum tested mesh size of 128 cells (along one dimension), the gap is reduced by up to 30 times.

5. Conclusion

Performance of low-level (CUDA) and high-level (OpenACC, OpenMP) programming models in various scenarios studied in this paper.

Evaluation of Tesla V100 memory bandwidth tests shows that OpenMP and OpenACC compilers are able to produce efficient parallel code in simple cases.

At the same time, as the complexity of code grows, we observe a significant performance gap between CUDA and OpenACC, OpenMP. In case of the sum reduction memory access pattern

OpenMP is 80% slower than the same optimized reduction pattern written in CUDA. The same trend observed while benchmarking Cloverleaf and LULESH applications.

The experimental results demonstrate that the performance of OpenACC and especially OpenMP codes may show degradation on large data inputs. It may indicate that the major target for optimization that could significantly speed-up OpenMP and OpenACC is compiler optimization.

Hand tuned CUDA matrix multiplication shows very promising results comparable to vendor-optimized cuBLAS both on Tesla V100 and MKL libraries on CPUs, but there is still a big amount of work on tuning for particular GPU architecture.

Acknowledgments

The study, excluding testing on GPU Nvidia Tesla V100, was partly supported by the Russian Science Foundation (project No. 20-71-10127). Experimental platform with Nvidia Tesla V100 GPUs is supported in part through computational resources of HPC facilities at NRU HSE.

References

[1] OpenMP Architecture Review Board. OpenMP Application Programming Interface. 2015. Ver 4.5.
[2] OpenACC-Standard.org. The OpenACC Application Programming Interface. 2018. Ver 2.7.
[3] Nvidia. CUDA Toolkit Documentation v10.2.89. 2019.
[4] The Khronos Group. The OpenCL(TM) Specification. 2019.
[5] Lindholm E, Nickolls J, Oberman S, Montrym J. 2008. *IEEE micro.* 39-55
[6] Volkov V, Demmel JW. 2008. *In SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing,* 1-11
[7] Stegailov V et al. 2017. *In International Conference on Parallel Processing and Applied Mathematics, Springer, Cham..* 327-336
[8] Stegailov V et al. 2019. International Journal of High Performance Computing Applications. 33(3):507-21.
[9] Deakin T et al. 2018. *International Journal of Computational Science and Engineering.* 17(3):247-62.
[10] Deakin T et al. 2016. *In International Conference on High Performance Computing, Springer, Cham..* 489-507
[11] Mallinson A C et al. 2013. *The Cray User Group.*
[12] Karlin I. 2012. *(LLNL), Livermore, CA (United States).*
[13] Memeti S et al. 2017. *In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing* 1-6
[14] Hoshino T, Maruyama N, Matsuoka S, Takaki R. 2013. *In 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing.* 136-143
[15] Guo X, Wu J, Wu Z, Huang B. 2016. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.* 9(4):1653-62.
[16] Wienke S, Terboven C, Beyer J C, Müller MS. 2014. *In European Conference on Parallel Processing, Springer, Cham.* 812-823
[17] Blackford L S et al. 2002. *ACM Transactions on Mathematical Software.* 28(2):135-51.