MSC 68U01 **DOI: 10.14529/mmp260108**

PECULIARITY OF DYNAMIC MEMORY ALLOCATION USING THE OPENACC STANDARD

N.M. Kuzmin[1]***, A.V. Khoperskov***[1]
[1]Volgograd State University, Volgograd, Russian Federation
E-mail: nikolay.kuzmin@volsu.ru, khoperskov@volsu.ru

The peculiarity of mapping the memory of the central processing unit (CPU) to the memory of the graphics processing unit (GPU) is discussed in the case of its dynamic allocation using OpenACC directives. The problem is that the array is not placed contiguously in memory. Therefore, the program associates all bytes of the computer's RAM between the start and end elements of the dynamic array with bytes of the GPU's memory, regardless of whether all of these bytes are actually occupied by the array elements. This leads to an unjustified and unpredictable increase in the size of the GPU memory allocated to store the dynamic array. Our simulations show that the increase in memory size can reach two orders of magnitude. The source code for dynamic allocation of a contiguous block of memory for two-dimensional arrays in C is given. This approach can be easily generalized to the case of an arbitrary number of dimensions. Testing of the described method showed that the sizes of dynamic arrays in the memory of the central and graphic processors coincide.

Keywords: dynamic arrays; memory allocation; OpenACC standard; parallel computing.

Introduction

A wide variety of mathematical modeling problems involve solving integral and/or differential equations, which requires the use of numerical methods. Grid-based computational algorithms imply discretization of space by a finite set of grid cells or nodes [1]. The accuracy of solving discrete analogues of the original continuous equations is determined by the number of cells and the principle "the finer the grid, the better the result". Therefore, it is necessary to perform a large number of identical calculations for each grid element, the efficiency of which is ensured by parallel computing, including on graphics processors.

There are several technologies to use the computing capabilities of GPUs [2]. OpenCL and CUDA require significant reworking or even a complete rewrite of the existing source code for the CPU. In addition, the second technology is limited to using only NVIDIA GPUs. OpenMP technology was originally designed for parallel computing in multiprocessor systems with shared memory, although recent versions allow the use of NVIDIA/AMD/Intel [3]. Therefore, if there is already a tested and debugged source code for computer simulation of some phenomenon or process, then the OpenACC standard is most convenient for performing calculations on the GPU [4, 5]. In fact, this standard extends the capabilities of the C/C++ and Fortran languages [6, 7]. It is based on the use of preprocessor directives similar to OpenMP directives, which allows you to create a new version of the program for execution on GPUs with minimal time costs and almost without changing the source code for CPU [3, 8]. The compiler ignores these directives when creating a sequential version of the program. In the following we use the terminology of the OpenACC standard, where the GPU is the "device" and the CPU is the "host".

The bottleneck in using GPUs to speed up computations is the data exchange between the host and the device. The standard strategy is to use only the device's memory whenever possible and minimize the data exchange between it and the host. The size of the GPU's RAM is usually significantly smaller than the CPUs in modern computer systems. Therefore, GPU memory is a more expensive resource in parallel simulations.

The sizes of the data arrays storing grid values are usually unknown at the stage of source code compilation. Therefore, dynamic memory allocation is used based on

the feature of the compiler from the NVIDIA HPC SDK package with support for the OpenACC standard, when data is effectively mapped from the host memory to the device memory only if they are contiguous in memory (Section 2). Ignoring this feature leads to the fact that the size of the memory allocated for data arrays on the device significantly exceeds the size actually occupied by the data (Section 3). We provide code examples that allow for the efficient allocation of multidimensional dynamic arrays in memory during computations on GPU using the OpenACC standard. The codes in this work are tested using the compiler from the NVIDIA HPC SDK version 25.5 on the Ubuntu 24.04 operating system on various computers with different processors and memory types.

1. Example of Parallelization Using Openacc Technology

We perform the analysis on the example of the numerical solution of the Laplace equation on a two-dimensional square Cartesian grid using the Jacobi iteration method. Listing 1 shows a short fragment of the source code using OpenACC directives in C.

```
double **a, **a_new;
int i, j, N, M, iter = 0;
const int iter_max = 1024;
/* Code for initializing variables is omitted */
/* Memory allocation on the host */
a     = allocate_2d_d(N, M);
a_new = allocate_2d_d(N, M);
/* Allocating memory on the device and copying array 'a' */
#pragma acc enter data create(a[0:N][0:M], a_new[0:N][0:M])
#pragma acc update device(a[0:N][0:M])
do {
   /*  Main computational kernel */
#pragma acc parallel loop copyin(N, M) present(a, a_new) \
collapse(2) independent
    for (i = 1; i < N-1; ++i)
        for (j = 1; j < M-1; ++j)
            a_new[i][j] = 0.25 * (a[i+1][j ] + a[i-1][j ]
                                + a[i ][j+1] + a[i ][j-1]);
        /* The calculation of the solution error and setting of
         boundary conditions is omitted for code compactness */
    iter = iter + 1;
} while (iter < iter_max);
```

Listing 1. Example of solution of 2D Laplace equation

The function `malloc` provides dynamic memory allocation in C programs. It receives the required number of bytes as an argument and, if successful, returns a pointer to the beginning of a free memory area of the appropriate size. Memory allocation for a 2D dynamic array is usually implemented as follows. Memory is allocated separately for each of its rows, and their addresses are stored in an array of pointers. Since accessing array elements using square brackets in C is syntactic sugar for address arithmetic, the above approach allows accessing array elements in the usual way using square brackets, in which the element indices are specified. The source code of the `allocate_2d_d` function in Listing 1 for dynamic memory allocation of a 2D array is shown in Listing 2. The check for successful memory allocation is omitted for code compactness.

```
double** allocate_2d_d(int N, int M) {
   int i;
   /* Allocating memory for the array of pointers to rows */
    a = (double **)malloc(N*sizeof(double *));
```

```
    for (i = 0; i < N; ++i)
        /* Allocate memory for rows.
           Their addresses are stored in the array of pointers */
        a[i] = (double *)malloc(M*sizeof(double));
    return a;
}
```

Listing 2. Memory allocation for 2D array in the usual way

2. Problem of Mapping Dynamic Arrays from Host Memory to Device Memory

Using the dynamic memory allocation method for a two-dimensional array (See Listing 2) results in the array rows almost never being stored contiguously in memory, as in static arrays (Figure 1 (a)).

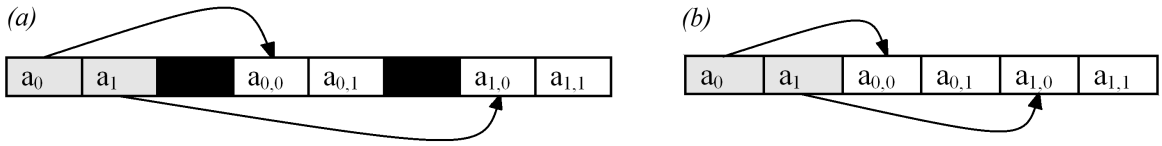

Fig. 1. The layout of the elements of the 2D array 2×2 in memory for the method in Listing 2 (a) and Listing 3 (b). The gray fill shows the location of pointers to array rows. The black color indicates memory areas inaccessible to the program. The arrows show the connections of pointers to rows with the beginning of rows

This leads to an unexpected problem in the case of the OpenACC technology in NVIDIA implementation. The size of the memory M_{GPU} occupied by the array on the GPU may be significantly larger than the size of the memory of this array on the host M_{host} (Fig. 2). This difference can reach $M_{GPU}/M_{host} \simeq 100$ according to the results of our computational experiments. The blue line shows the initial data size M_{host} as a function of the number of array cells N_{tot}. We compute the allocated memory size on the GPU M_{GPU} [bytes] in 192 numerical experiments with different N_{tot} on several computers at different points in time. All our simulation results lie inside the gray area, where the red line bounds the size of M_{GPU} in our sample of numerical experiments. The red dashed line shows the possible upper limit of the memory size allocated on the device. Memory allocation may not be repeated for different program runs, since the result depends on the processes executed by the operating system before starting the application program.

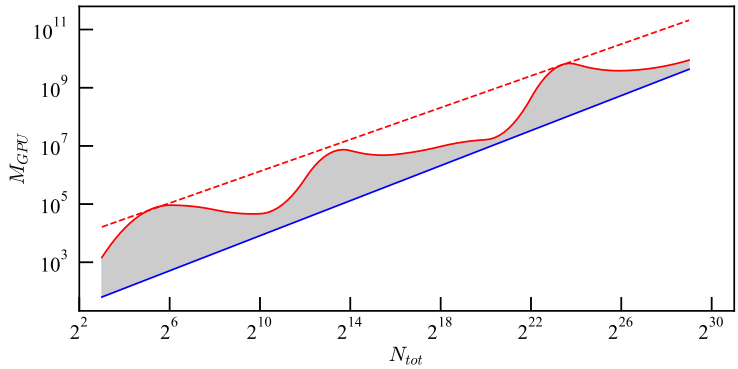

Fig. 2. Allocated memory on GPU for storing arrays with given number of grid cells for different program runs

The above problem is related to the specifics of the directive execution `#pragma acc enter data create (a[0:N][0:M])`. The OS in runtime determines the minimum and maximum addresses of the array elements dynamically allocated in the host

memory. Then the entire memory size between these addresses is mapped from the host to the device. This happens even if not all bytes from this memory size are actually occupied by the array elements. Our analysis shows that this happens almost always in the case of the code from Listing 2 (See Figure 1 *a*). We emphasize that it is impossible to calculate in advance the size of memory on the device mapped with the array on the host. The result of dynamic memory allocation depends on the current state of memory, which in modern multitasking and multi-user OSs depends on many external uncontrollable factors.

Another interesting point is that the array rows can be randomly located in memory when allocating memory using the code in Listing 2, without taking into account the ascending order of their numbers. It is important to emphasize that the official NVIDIA HPC SDK guide [9] states that arrays in host memory, mapped to arrays in device memory, must be contiguous in memory.

The above problem of memory mapping can be solved by using 1D arrays instead of 2D arrays with a special indexing method in the form of the construction `a[i*M + j]` instead of the usual notation `a[i][j]`. However, this solution leads to cumbersome and poorly readable code, especially for arrays with dimensions greater than two, which is a potential source of subtle errors in the program. A more preferable method is to allocate memory for the entire data size of the multidimensional array at once with the correct setting of pointers to the rows within this memory. An example of such an approach is shown in Listing 3, where the check for successful memory allocation is omitted for code compactness. We use contiguous array placement in memory (See Fig. 1 (*b*) without black rectangles between memory sections, See Fig. 1 (*a*)).

```
double** allocate_2d_d(int N, int M) {
    /*Pointers to the the array and the its data beginning*/
     double **a, *data;
     int i;
    /* Allocate memory for rows and pointers to them */
     a = (double **)malloc(N*sizeof(double *) +
          N*M*sizeof(double));
    /* Set 'data' pointer to the beginning of the array data
          (skip the memory section with pointers to rows) */
     data = (double *)(a + N);
     for (i = 0; i < N; ++i)
          /* Setting up array of rows pointers */
     a[i] = (double *)(data + i*M);
     return a;
}
```

Listing 3. Dynamic memory allocation for 2D array in a continuous block

The approach in Listing 3 can be easily transferred to the case of n-dimensional arrays with $n \geq 3$. Note also that the above-mentioned feature of dynamic memory allocation is absent, for example, in modern versions of the Fortran language using the operator `ALLOCATE`, which is passed the number of array dimensions and their size when called. Therefore, its algorithm is similar to the algorithm for the C language, presented in Listing 3.

Conclusion

We analyzed the dependence of the allocated memory size for multidimensional dynamic arrays on the GPU on the continuity of the array elements' location in memory. The results show that the absence of this continuity can lead to an increase in the corresponding memory size on the GPU by two orders of magnitude.

The OpenACC technology for creating a program for GPU based on the original program for sequential execution on CPU is simple, but requires taking into account the compiler features. The considered method of dynamic memory allocation for two-

dimensional arrays can easily be generalized to the case of a larger number of dimensions. We limit ourselves to the simplest method of memory allocation (Listing 2), which is often found in various codes, especially in the presence of a large number of multidimensional arrays. Examples are simulations of hydro- and MHD-flows, plasma dynamics in different approximations, deformations of elastic solids, etc. The literature recommends avoiding either multidimensionality or dynamic memory when porting code to a GPU. However, using one-dimensional arrays with a special indexing method (See Section 2) makes the source code difficult to read and is a potential source of subtle bugs.

Acknowledgements. *This work is supported by the Russian Science Foundation (grant no. 23-71-00016, https://rscf.ru/project/23-71-00016/.*

References

1. Gervich L.R., Steinberg B.Ya. Automation of the Application of Data Distribution with Overlapping in Distributed Memory. *Bulletin of the South Ural State University. Series: Mathematical Modelling, Programming and Computer Software*, 2023, vol. 16, no. 1, pp. 59–68. DOI: 10.14529/mmp230105
2. Krasnov M.M., Feodoritova O.B. The Use of Functional Programming Library for Parallel Computing on CUDA. *Programming and Computer Software*, 2024, vol. 50, no. 1, pp. 11–23. DOI: 10.1134/S0361768824010055
3. Yohei Miki, Toshihiro Hanawa. Unified Schemes for Directive-Based GPU Offloading. *IEEE Access*, 2024, vol. 12, no. 1, pp. 181644–181665. DOI: 10.1109/ACCESS.2024.3509380
4. *OpenACC-Standard.org. The OpenACC Application Programming Interface Version 3.4*. Available at: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.4.pdf (accessed on 15.01.2026).
5. Subramanian S., Balsara D.S., Bhoriya D. et al. Techniques, Tricks, and Algorithms for Efficient GPU-Based Processing of Higher Order Hyperbolic PDEs. *Communications on Applied Mathematics and Computation*, 2024, vol. 6, no. 4, p. 2336–2384. DOI: 10.1007/s42967-022-00235-9
6. Tomanovic I., Belosevic S., Milicevic A. et al. CFD Code Parallelization on GPU and the Code Portability. *Advanced Theory and Simulations*, 2025, vol. 8, no. 3, article ID: 2400629. DOI: 10.1002/adts.202400629
7. Joel E. D., Seyong Lee, Valero-Lara P. et al. Clacc: OpenACC for C/C++ in Clang. *The International Journal of High Performance Computing Applications*, 2024, vol. 38, no. 5, pp. 427–446. DOI: 10.1177/10943420241261976
8. Krasnov M.M., Feodoritova O.B. Functional Programming Libraries for Graphics Accelerators. *Supercomputing Frontiers and Innovations*, 2022, vol. 9, no. 4, pp. 28–37. DOI: 10.14529/jsfi220403
9. *NVIDIA Corporation. NVIDIA HPC SDK Version 25.5 Documentation*. Available at: https://docs.nvidia.com/hpc-sdk/index.html (accessed on 15.01.2026).

Received September 19, 2025

УДК 519.6 **DOI: 10.14529/mmp260108**

ОСОБЕННОСТЬ ДИНАМИЧЕСКОГО ВЫДЕЛЕНИЯ ПАМЯТИ ПРИ ИСПОЛЬЗОВАНИИ СТАНДАРТА OPENACC

Н.М. Кузьмин[1]***, А.В. Хоперсков***[1]
[1]Волгоградский государственный университет, г. Волгоград, Российская Федерация

Особенность отображения памяти центрального процессора (CPU) в память графического процессора (GPU) обсуждается при ее динамическом выделении с исполь-

зованием директив OpenACC. Проблема состоит в том, что программа ассоциирует все байты оперативной памяти компьютера между начальным и конечным элементами динамического массива с байтами памяти графического процессора вне зависимости от того, все ли эти байты в действительности заняты элементами массива. Это происходит в случае, если данные не занимают единое (непрерывное) пространство в оперативной памяти. Это приводит к неоправданному и непрогнозируемому заранее увеличению объема памяти графического процессора в десятки и сотни раз, которое выделяется для хранения динамического массива. Приведен исходный код динамического выделения непрерывного блока памяти для двумерных массивов на языке C. Данный подход может быть легко обобщен на случай большего количества измерений. Проведенное тестирование показало, что при таком подходе размеры динамических массивов в памяти центрального и графического процессоров совпадают.

Ключевые слова: динамические массивы; выделение памяти; стандарт OpenACC; параллельные вычисления.

Литература

1. Гервич, Л.Р. Об автоматизации применения размещения данных с перекрытиями в распределенной памяти / Л.Р. Гервич, Б.Я. Штейнберг // Вестник ЮУрГУ. Серия: Математическое моделирование и программирование. – 2023. – Т. 16, № 1. – С. 59–68.
2. Krasnov, M.M. The Use of Functional Programming Library for Parallel Computing on CUDA / M.M. Krasnov, O.B. Feodoritova // Programming and Computer Software. – 2024. – V. 50, № 1. – P. 11–23.
3. Yohei Miki. Unified Schemes for Directive-Based GPU Offloading / Yohei Miki, Toshihiro Hanawa // IEEE Access. – 2024. – V. 12, № 1. – P. 181644–181665.
4. OpenACC-Standard.org. The OpenACC Application Programming Interface Version 3.4. – URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.4.pdf (дата обращения: 15.01.2026).
5. Subramanian, S. Techniques, Tricks, and Algorithms for Efficient GPU-Based Processing of Higher Order Hyperbolic PDEs / S. Subramanian, D.S. Balsara, D. Bhoriya et al // Communications on Applied Mathematics and Computation. – 2024. – V. 6, № 4. – P. 2336–2384.
6. Tomanovic, I. CFD Code Parallelization on GPU and the Code Portability / I. Tomanovic, S. Belosevic, A. Milicevic et al // Advanced Theory and Simulations. – 2024. – V. 8, № 3. – Article ID: 2400629.
7. Joel, E. D. Clacc: OpenACC for C/C++ in Clang. / E. D. Joel, Seyong Lee, Valero-Lara P. et al // The International Journal of High Performance Computing Applications. – 2024. – V. 38, № 5. – P. 427–446.
8. Krasnov, M.M. Functional Programming Libraries for Graphics Accelerators / M.M. Krasnov, O.B. Feodoritova // Supercomputing Frontiers and Innovations. – 2022. – V. 9, № 4. – P. 28–37.
9. NVIDIA Corporation. NVIDIA HPC SDK Version 25.5 Documentation. – URL: https://docs.nvidia.com/hpc-sdk/index.html (дата обращения: 15.01.2026).

Николай Михайлович Кузьмин, кандидат физико-математических наук, доцент, кафедра информационных систем и компьютерного моделирования, Волгоградский государственный университет (г. Волгоград, Российская Федерация), nikolay.kuzmin@volsu.ru.

Александр Валентинович Хоперсков, доктор физико-математических наук, профессор, кафедра информационных систем и компьютерного моделирования, Волгоградский государственный университет (г. Волгоград, Российская Федерация), khoperskov@volsu.ru.

Поступила в редакцию 19 сентября 2025 г.