2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

doi 10.26089/NumMet.v27r321

Selection of the most efficient sorting algorithm for small-sized data structures by an integer field

Andrey A. Galyuzov
All-Russian Research Institute of Automation named after N. L. Dukhov (VNIIA), Moscow, Russia
ORCID: 0009-0002-0031-8433, e-mail: AAGalyuzov@vniia.ru

Abstract: The problem of repeatedly sorting an array of movable particles by an integer field that specifies the type of physical interaction is considered. The study compares 16 sorting implementations for an array of $2 \cdot 10^7$ structures on CPUs and a GPU. The optimal algorithm is defined as the one with the lowest average execution time on fixed input data and a chosen computing platform; when average execution times are close, a smaller amount of auxiliary memory is considered an additional advantage. All algorithms were tested on the same input array, where the integer sorting key took random values from the set $\{-1, 0, 1, 2, 3\}$. For the GPU, only *on-device* sorting time measured using CUDA events was compared; memory allocation, data transfers between host and device, warm-up, and other runtime overheads were not included in the measurement interval. It is shown that for the considered problem, the comparison-based sorting algorithms and integer-key sorting methods are fundamentally different: due to the small number of possible sorting key values, specialized methods such as counting sort and radix sort are significantly more efficient than general-purpose comparison algorithms. On CPUs, the best results were achieved by the custom specialized algorithm TPT3 and schemes with parallel sorting of subarrays, while on an NVIDIA Tesla V100 GPU, the leader was *DeviceRadixSort* from the CUDA CUB library. The conclusions drawn apply to the specific movable particle structure and key range, the *on-device* execution scenario on a GPU and shared-memory computing systems.

Keywords: parallel computations, radix sort, counting sort, OpenMP, CUDA, CUDA Thrust, CUDA CUB, CPU, GPU.

For citation: A. A. Galyuzov, "Selection of the most efficient sorting algorithm for small-sized data structures by an integer field," Numerical Methods and Programming. **27** (3), 317–333 (2026). doi 10.26089/NumMet.v27r321.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

Выбор наиболее производительного алгоритма сортировки структур данных малого размера по целочисленному полю

А. А. Галюзов
Всероссийский научно-исследовательский институт автоматики имени Н. Л. Духова (ВНИИА), Москва, Российская Федерация
ORCID: 0009-0002-0031-8433, e-mail: AAGalyuzov@vniia.ru

Аннотация: Рассматривается задача многократной сортировки массива переносимых частиц по целочисленному полю, задающему тип физического взаимодействия. Сравниваются 16 реализаций сортировки массива из $2 \cdot 10^7$ структур на центральных процессорах и графическом ускорителе. Под оптимальным понимается алгоритм с наименьшим средним временем выполнения на фиксированных входных данных и выбранной вычислительной платформе; при близких средних значениях времени выполнения дополнительным преимуществом считается меньший объем вспомогательной памяти. Для всех алгоритмов использовался один и тот же входной массив, в котором целочисленный ключ сортировки принимал случайные значения из множества $\{-1, 0, 1, 2, 3\}$. Для GPU сравнивалось только *on-device* время сортировки, измеренное при помощи CUDA events; выделение памяти, переносы данных между host и device, warm-up и прочие накладные расходы среды выполнения в интервал измерения не включались. Показано, что для рассматриваемой задачи алгоритмы сортировок, основанные на сравнениях, и методы сортировки целочисленных ключей принципиально различаются: из-за малого числа возможных значений ключа сортировки специализированные методы, такие как сортировка подсчетом и поразрядная сортировка, оказываются существенно эффективнее универсальных алгоритмов сравнения. На центральных процессорах лучшие результаты показали специализированный авторский алгоритм TPT3 и схемы с параллельной сортировкой подмассивов, а на GPU NVIDIA Tesla V100 лидером стала *DeviceRadixSort* из библиотеки CUDA CUB. Полученные выводы относятся к рассматриваемым структуре переносимой частицы и диапазону ключа, *on-device* сценарию выполнения на GPU, а также вычислительным системам с общей оперативной памятью.

Ключевые слова: параллельные вычисления, поразрядная сортировка, сортировка подсчетом, OpenMP, CUDA, CUDA Thrust, CUDA CUB, CPU, GPU.

Для цитирования: Галюзов А.А. Выбор наиболее производительного алгоритма сортировки структур данных малого размера по целочисленному полю // Вычислительные методы и программирование. 2026. **27**, № 3. 317–333. doi 10.26089/NumMet.v27r321.

1. Introduction. Problems involving sorting large arrays of small data structures by a short integer key regularly arise in particle transport simulation codes, where objects must be grouped by physical interaction type, state, or another discrete characteristic [1–3]. In addition, sorting simulated particles located in the same cell of a computational domain by various parameters in order to reduce the number of cache misses is also used, for example, in molecular dynamics, particle-in-cell (PIC) and hydrodynamics codes [4–7]. In such problems, the choice of sorting method affects not only the time spent on ordering itself, but also data locality and, consequently, the efficiency of subsequent computational stages.

In the general-purpose Geant4 toolkit [8, 9] and in specialized radiation transport codes, the task of grouping particles by interaction type arises naturally. In TPT3 (Toolkit for Particle Transport 3, a framework for particle transport simulation, version 3), a high-performance parallel Monte Carlo radiation transport simulation code [10, 11], which runs on CPUs using OpenMP [12] and on GPUs using CUDA [13] and OpenACC [14], such regrouping is used as a part of the computational pipeline and directly affects the performance of parallel computations. It is needed to improve locality of data of the same type, which is required for their efficient parallel processing and for reducing the number of cache misses.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

It is important to emphasize that the question of the "best" sorting algorithm is meaningful only after the problem model has been specified. For comparison-based sorts, the lower bound on the number of comparisons is $O(n \log n)$ [15, 16]. At the same time, for integer keys with a bounded range, methods based on counting, digit-wise processing or bucket distribution can be linear in n [16–18]. Therefore, there is no universal algorithm that is equally optimal for all problem statements: the choice is determined by the key range, the size of the transported particle structure, stability requirements, available memory and the particular computing platform.

This distinction is clearly visible in general-purpose library implementations. On CPUs, hybrid schemes of the *introsort* family are often used as baseline algorithms [19], whereas specialized hybrid methods, such as *spreadsort* [17], are used for integer keys. On GPUs, variants of radix sorting and key-based sorting aimed at high memory throughput have become widespread [18].

In TPT3, an array of transported particle structures is sorted by an integer field `ir`, which denotes a type of physical interaction. In the current problem statement, the field `ir` takes only five values, from -1 to 3. For this reason, the comparison between general-purpose comparison-based sorting algorithms and specialized integer-key methods is especially illustrative here.

In this paper, an *optimal* algorithm is understood as the algorithm with the smallest average execution time for fixed input data on a particular shared-memory computing platform. When average times are close, a smaller amount of additional memory is treated as an advantage. Sorting stability, i.e. preservation of the relative order of elements with equal key values, is considered an important but not primary characteristic.

The purpose of this work is to compare 16 implementations for sorting a particle array in TPT3 on CPUs and a GPU, to determine the most efficient variants for the considered particle structure and to indicate the limits within which the obtained conclusions are applicable. The work does not claim to find a universally best sorting algorithm. The analysis concerns sorting an array of structures of the form shown in Listing 1 by a low-cardinality integer key `ir` on a particular shared-memory computing platform.

2. Problem statement and comparison criteria. In the test problem, an array of $N_\mathrm{p} = 2 \cdot 10^7$ transported particle structures is sorted, which requires approximately 1 GB of memory on the computing device. The input array size was chosen as the largest one for which the memory consumption even of the most resource intensive algorithms fits into the 8 GB RAM of the computer with an Intel Core i7-3770 CPU used in the calculations. The particle structure is described in Listing 1, followed by the declarations of the input

Listing 1. The minimal structure describing a transported particle

```
typedef std::array<double, 3> double3;
struct P{
int ir;
int id;
double3 r;
double3 p;
P():ir(-100),id(-1),r{},p{}{}
P(int _ir, int _id):ir(_ir),id(_id),r{},p{}{}
operator unsigned int() const
{ return static_cast<unsigned int> (ir) ^ 0x80000000; }
bool operator>(const P & rhs) const
{ return (ir>rhs.ir); }
bool operator<(const P & rhs) const
{ return (ir<rhs.ir); }
bool operator==(const P & rhs) const
{ return (ir==rhs.ir); }
bool operator!=(const P & rhs) const
{ return (ir!=rhs.ir); }
};

vector<P> particles(N);
vector<P> particles_out(N);
```

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

and output particle arrays. Below, the neutral term "structure" is used rather than POD structure, because the specified data type contains user-defined constructors and overloaded operators.

The integer field `ir` is used as the sorting key. In the current version of TPT3, it takes values from the set $\{-1, 0, 1, 2, 3\}$; hence the number of distinct keys K is equal to 5. For such a small key range, counting-sort and radix-sort methods should be expected to outperform general-purpose comparison-based algorithms.

For all considered implementations, the same input array of N_p structures was used. The values of the field `ir` in the input array were generated randomly and uniformly over the specified range. The remaining fields of the structure were moved together with it as associated data. Before each sorting run, the same pre-generated copy of this array was supplied as input, which made it possible to compare average execution times of different algorithms directly.

Although sorting stability was not the main criterion for selecting an algorithm, this characteristic is useful in several applied and debugging scenarios. For example, when analyzing and verifying the computations described in [20], preserving the relative order of neutrons with the same interaction type substantially simplified error localization in the software implementation of the weighted modeling algorithm for the nuclear fission chain reaction.

3. Measurement methodology. A set of 16 sorting implementations was considered. The average execution time, sorting stability and additional memory requirements were compared. The main optimality criterion was the average execution time. The minimum amount of auxiliary memory was used as an additional criterion when average execution times were close.

Each computational task was run $m = 3, 4, 5$ times after a preliminary warm-up of the computing device, i.e. after several test executions of the same task to bring the processor and cache memory to an efficient intensive operating regime. For the individual run times $t_1, \ldots, t_m$, the sample mean

$$\bar{t} = \frac{1}{m} \sum_{i=1}^{m} t_i$$

and the sample standard deviation

$$s = \sqrt{\frac{1}{m-1} \sum_{i=1}^{m} \left(t_i - \bar{t}\right)^2}$$

were computed. The half-width of a two-sided 95% confidence interval for the mean execution time of the program code was determined by the formula

$$\Delta t = t_{0.975,m-1} \frac{s}{\sqrt{m}},$$

where $t_{0.975,m-1}$ is the quantile of Student's distribution with $m-1$ degrees of freedom. Table 1 gives the average times $\bar{t}$ rounded with regard to Δt, in order to prevent the presentation of the results from being overloaded with the full set of statistical data.

On the GPU, only the *on-device* sorting time was measured. It was determined using CUDA events placed immediately before the sorting call and immediately after it. Therefore, memory allocation, data transfers between the CPU (host) and GPU (device), warm-up, auxiliary GPU synchronization outside the measured interval and other runtime overheads were not included in the measured time. Thus, GPU results should be interpreted as results for a scenario in which the data are already located in device memory.

In Table 1, the first column gives the algorithm number, which coincides with the sequential number of the corresponding subsection in Section 4 of this paper. The second column indicates whether the relative order of particles with the same value of `ir` is preserved. For stable algorithms, this is achieved by the sorting property itself. For some comparison-based algorithms, the same effect was achieved by additional ordering with respect to the field `id`. The third column gives the number of subarrays N_bin, and the columns R(`-O0`) and R(`-O3`) show the ratio of the average execution time of the corresponding sorting algorithm variant to the average execution time of algorithm 4.16 at the same compiler optimization level. This relative scale is used only as a compact way of presenting the results within the considered set of numerical experiments and is not intended for a direct architectural comparison between CPUs and GPUs.

The GitHub repository with the source code for the examples from this article is located at [21]. The files mentioned below are contained in that repository.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

Table 1. The table of comparative performance of different programming algorithms of sorting of an array of $2 \cdot 10^7$ transported particle structures on different computing platforms. The algorithm number in the first column matches the sequential number of the subsection in Section 4 of this paper

No.	Order	N_{bin}	$\bar{t}$, ms (`-O0`)	R(`-O0`)	$\bar{t}$, ms (`-O3`)	R(`-O3`)	Platform
1	no	no	5700	×211	1040	×43.3	4-core CPU Intel Core i7-3770
1	yes	no	7800	×289	2100	×87.5	4-core CPU Intel Core i7-3770
1	no	no	5300	×196	980	×40.8	64-core CPU Intel Xeon Gold 6242
1	yes	no	7200	×267	2040	×85	64-core CPU Intel Xeon Gold 6242
1	no	no	5300	×196	950	×39.6	72-core CPU Intel Xeon E5-2697
1	yes	no	7900	×293	2100	×87.5	72-core CPU Intel Xeon E5-2697
2	no	no	1100	×40.7	270	×11.3	4-core CPU Intel Core i7-3770
2	no	no	1000	×37	270	×11.3	64-core CPU Intel Xeon Gold 6242
2	no	no	1200	×44.4	340	×14.2	72-core Intel Xeon E5-2697
3	no	no	340	×12.6	340	×14.2	GPU NVIDIA Tesla V100
4	no	no	2100	×77.8	2000	×83.3	4-core CPU Intel Core i7-3770
4	no	no	2700	×100	2500	×104.2	64-core CPU Intel Xeon Gold 6242
4	no	no	2500	×92.6	2300	×95.8	72-core CPU Intel Xeon E5-2697
5	yes	4	150	×5.6	140	×5.8	4-core CPU Intel Core i7-3770
5	yes	64	650	×24.1	530	×22.1	64-core CPU Intel Xeon Gold 6242
5	yes	2048	85	×3.1	80	×3.3	64-core CPU Intel Xeon Gold 6242
5	yes	72	520	×19.3	470	×19.6	72-core CPU Intel Xeon E5-2697
5	yes	2048	93	×3.4	90	×3.8	72-core CPU Intel Xeon E5-2697
6	yes	8192	41	×1.5	41	×1.7	GPU NVIDIA Tesla V100
7	no	no	91	×3.4	93	×3.9	GPU NVIDIA Tesla V100
8	yes	4	2000	×74.1	760	×31.7	4-core CPU Intel Core i7-3770
8	yes	512	1390	×51.5	255	×10.6	4-core CPU Intel Core i7-3770
8	yes	64	340	×12.6	130	×5.4	64-core CPU Intel Xeon Gold 6242
8	yes	512	273	×10.1	94	×3.9	64-core CPU Intel Xeon Gold 6242
8	yes	72	320	×11.9	270	×11.3	72-core CPU Intel Xeon E5-2697
8	yes	512	280	×10.4	120	×5	72-core CPU Intel Xeon E5-2697
9	yes	4	1500	×55.6	700	×29.2	4-core CPU Intel Core i7-3770
9	yes	2048	830	×30.7	210	×8.8	4-core CPU Intel Core i7-3770
9	yes	64	250	×9.3	120	×5	64-core CPU Intel Xeon Gold 6242
9	yes	2048	180	×6.7	92	×3.8	64-core CPU Intel Xeon Gold 6242
9	yes	72	270	×10	220	×9.2	72-core CPU Intel Xeon E5-2697
9	yes	2048	204	×7.6	120	×5	72-core CPU Intel Xeon E5-2697
10	no	256	180	×6.7	180	×7.5	GPU NVIDIA Tesla V100
11	no	4096	45000	×1670	2900	×121	4-core CPU Intel Core i7-3770
11	no	4096	3300	×122	410	×17.1	64-core CPU Intel Xeon Gold 6242
11	no	4096	4600	×170.4	490	×20.4	72-core CPU Intel Xeon E5-2697
12	no	4	830	×30.7	550	×22.9	4-core CPU Intel Core i7-3770
12	no	64	940	×34.8	680	×28.3	64-core CPU Intel Xeon Gold 6242
12	no	72	920	×34.1	590	×24.6	72-core CPU Intel Xeon E5-2697
13	no	4096	220	×8.1	220	×9.2	GPU NVIDIA Tesla V100
14*	no	1024	770	×28.5	810	×33.8	GPU NVIDIA Tesla V100
15	no	1024	120	×4.4	120	×5	GPU NVIDIA Tesla V100
16	no	no	27	×1	24	×1	GPU NVIDIA Tesla V100

* The average time for algorithm 4.14 was obtained by extrapolating measurements on smaller arrays; therefore, this result should be regarded as indicative

Table 2 shows the main characteristics of the used computing platforms. For the Intel Xeon Gold 6242 and Intel Xeon E5-2697 CPUs, the table lists the characteristics of a single processor, whereas two-socket compute nodes were used in the experiments. Therefore, below, the designations “64-core Intel Xeon Gold 6242” and “72-core Intel Xeon E5-2697” refer to the complete compute node and mean, respectively, 64 and 72 logical cores of two processors. For the GPU, the values in the columns “Number of cores” and “Number of threads” are given in the terminology of the device specification and are not used as an independent basis for quantitative comparison with CPUs. The paper compares fixed hardware configurations. A direct comparison by transistor count or die area was not used as an independent criterion for explaining relative performance.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

Table 2. Comparative characteristics of used computing platforms

Platform	Number of cores	Number of threads	Basic frequency, GHz	Technology, nm	Chip size, mm^2	Number of gates, billion
CPU Intel Core i7-3770	4	4	3.4	22	160	1.4
CPU Intel Xeon Gold 6242	16	32	2.8	14	484	∼7
CPU Intel Xeon E5-2697	18	36	2.3	14	456.1	7.2
GPU NVIDIA Tesla V100	5120 CUDA cores	5120	1.4	12	815	21.1

4. Results of numerical experiments.

4.1. Standard C++ sorting with `std::sort`. In this paper, `std::sort` is used as a comparison-based baseline general-purpose sorting algorithm (Listing 2). In typical standard library implementations, it belongs to hybrid algorithms of the `introsort` family [19], but the C++ standard does not prescribe a particular internal implementation. Therefore, here we are primarily interested in its performance on the selected platforms.

Listing 2. Usage of a C++ standard sorting algorithm `std::sort`

```
1 std::sort(particles.begin(),particles.end(),
2          [](const P & l, const P & r)->bool {return l.ir>r.ir;});
```

For the considered problem, `std::sort` is inferior to methods that exploit the small range of the key `ir`. On the 4-core Intel Core i7-3770 CPU, the average execution time without preserving the order of particles with equal `ir` is 1.04 s at `-O3` and 5.7 s at `-O0`. When the order preservation requirement is added, the corresponding times are 2.1 s and 7.8 s. On the 64-core Intel Xeon Gold 6242, the times without order preservation are 980 ms at `-O3` and 5.3 s at `-O0`, and on the 72-core Intel Xeon E5-2697 they are 950 ms and 5.3 s, respectively.

Sorting without order preservation is defined by the comparator `l.ir > r.ir`. For the variant in which a reproducible ordering of particles with the same `ir` by increasing `id` was required for certain tasks, a secondary ordering by `id` was added to the comparator:

```
l.ir > r.ir || (l.ir == r.ir && l.id < r.id).
```

Under the assumption that the field `id` monotonically corresponds to increasing particle age, this sorting mode preserves the required relative order of elements within particle groups with the same `ir`.

To avoid overloading the subsequent presentation, for unstable algorithms only the variant without preserving the relative order of particles with the same interaction type is discussed separately below.

4.2. Sorting from the Boost Sort library. Numerical experiments showed that, among Boost Sort algorithms, the `boost::sort::spreadsort` family gives the best result for the considered problem. This algorithm combines ideas of radix sorting and comparison-based sorting [17], and its library implementation is described in [22]. Because sorting is performed by an integer field, the `boost::sort::spreadsort::integer_sort` variant was used. Its usage is shown in Listing 3.

On the 4-core Intel Core i7-3770 CPU and on the 64-core Intel Xeon Gold 6242, the average execution times were 270 ms at `-O3` and, respectively, 1.1 s and 1 s at `-O0`. On the 72-core Intel Xeon E5-2697, the times were about 340 ms at `-O3` and 1.2 s at `-O0`. Thus, `spreadsort` is noticeably faster than the standard

Listing 3. Usage of an algorithm `integer_sort` for integer sorting from the high-performance complex sorting approach `spreadsort` from the Boost Sort library

```
1 #include <boost/sort/spreadsort/spreadsort.hpp>
2 ...
3 boost::sort::spreadsort::integer_sort(particles.begin(), particles.end(),
4                                        [](const P & p, size_t offset) ->
5 unsigned int {return static_cast<unsigned int> (p.ir) ^ 0x80000000;});
```

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

comparison-based sort and serves as a strong library baseline for CPUs. However, in the considered problem statement it is still inferior to more specialized schemes.

4.3. Standard sorting and GPU offloading using an execution policy with the NVIDIA HPC SDK compiler. This variant is of interest as the least labor intensive way to move sorting to a GPU almost without changing the original C++ code. Listing 4 shows the use of the `std::execution::par` execution policy and the `-stdpar=gpu` flag of the `nvc++` compiler. Only the *on-device* time was included in the comparison, so this result should be interpreted specifically as an estimate of the efficiency of a library implementation of offloading to the graphics accelerator in the selected environment.

Listing 4. Usage of standard sorting and GPU offloading with the help of execution policy within the NVIDIA HPC SDK compiler

```
#include <algorithm>
#include <execution>
...
std::sort(std::execution::par, particles.begin(), particles.end());
```

On a server node with a 64-core Intel Xeon Gold 6242 CPU and an NVIDIA Tesla V100 GPU, this variant ran in 340 ms. Within the measurement accuracy, no dependence on the `-O0` or `-O3` compilation flags was observed. This approach is convenient because it does not require an explicit CUDA implementation, but in terms of performance it is inferior to specialized GPU methods.

4.4. A standard radix-sort algorithm. The file `radixsort.cpp` in the repository [21] contains an implementation of a standard radix sort on the CPU. Theoretically, radix sort is well suited to sorting by an integer key [16, 18], but the practical result is determined by implementation details, the nature of the data being moved and memory organization.

In the considered implementation, radix sorting of the full array of structures turned out to be substantially slower than the better performing variants. On the 4-core Intel Core i7-3770 CPU, the average execution time is 2 s at `-O3` and 2.1 s at `-O0`. On the 64-core Intel Xeon Gold 6242, the times were 2.5 s and 2.7 s, and on the 72-core Intel Xeon E5-2697 they were 2.3 s and 2.5 s, respectively. Therefore, merely belonging to the class of radix-sort algorithms does not guarantee high performance. The specific implementation plays the decisive role.

4.5. The specialized author-developed particle sorting algorithm in TPT3. Listing 5 and the file `benchmark.cpp` in the repository [21] present the specialized algorithm developed for TPT3. In essence, it is an extended and upgraded counting-sort method. First, the original array is divided into N_{bin} subarrays. Then, for each subarray, the number of particles with each value of `ir` is counted. Next, prefix offsets are computed. Finally, elements are copied to the output array into the required ranges.

Listing 5. The specialized TPT3 particle-structure sorting algorithm

```
template <size_t Nt>
void mysort_Nthreads(vector<P> & particles)
{
  int COUNTbin[5][Nbin];
  memset(COUNTbin,0,sizeof(COUNTbin));
  int OFFSET[5][Nbin];
  int COUNT[5];
  memset(COUNT,0,sizeof(COUNT));
  int COUNTERbin[5][Nbin];
  memset(COUNTERbin,0,sizeof(COUNTERbin));
  int init[Nbin];
  int fin[Nbin];
  const int dL=LIFE/Nbin;
  const int DL=dL+1;
  const int n=Nbin-LIFE%Nbin;
  #pragma omp parallel for
  for(int b=0; b<Nbin; b++)
```

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

```
  {
    if(b<n)
    {
      init[b]=b*dL;
      fin[b]=(b+1)*dL;
    }
    else if(b==n)
    {
      init[b]=n*dL;
      fin[b]=n*dL+DL;
    }
    else if(b>n)
    {
      init[b]=n*dL+DL*(b-n);
      fin[b]=n*dL+DL*(b-n+1);
    }
  }
  #pragma omp parallel for
  for(int b=0; b<Nbin; b++)
    for(int j=init[b]; j<fin[b]; j++)
      COUNTbin[(3-particles[j].ir)][b]++;
  #pragma omp parallel for
  for(int j=0; j<5; j++)
    for(int b=0; b<Nbin; b++)
      COUNT[j] += COUNTbin[j][b];
  OFFSET[0][0]=0;
  for(int j=1; j<5; ++j)
    OFFSET[j][0]=OFFSET[j-1][0]+COUNT[j-1];
  #pragma omp parallel for
  for(int j=0; j<5; j++)
    for(int b=1; b<Nbin; b++)
      OFFSET[j][b]=COUNTbin[j][b-1]+OFFSET[j][b-1];
  #pragma omp parallel for
  for(int b=0; b<Nbin; b++)
  {
    for(int j=init[b]; j<fin[b]; j++)
    {
      const int ir=3-particles[j].ir;
      particles_out[ OFFSET[ir][b] + COUNTERbin[ir][b]++ ]=particles[j];
    }
  }
}
```

For a fixed number of distinct keys $K = 5$, the computational complexity of this approach is $O(N_\mathrm{p})$, and the additional memory is $O(N_\mathrm{p} + K\,N_\mathrm{bin})$ because of the output array and auxiliary counters. The algorithm preserves the relative order of particles with the same value of `ir`, because within each subarray copying to the output array is performed in the original order. In this sense, it is a specialized stable variant of the counting-sort idea [16].

On the 4-core Intel Core i7-3770 CPU, the average execution time is about 140 ms at `-O3` and 150 ms at `-O0` in the case when $N_\mathrm{bin} = 4$ is chosen. On the 64-core Intel Xeon Gold 6242 CPU with $N_\mathrm{bin} = 64$, the times were 530 ms at `-O3` and about 650 ms at `-O0`; on the 72-core Intel Xeon E5-2697 CPU with $N_\mathrm{bin} = 72$, the times were approximately 470–520 ms.

The number of subarrays N_bin was standardly chosen to be equal to the number of logical cores of the corresponding CPU. However, it was found that at $N_\mathrm{bin} = 2048$ substantially higher performance can be achieved on many-core CPUs: 80 ms on the 64-core Intel Xeon Gold 6242 CPU and 90 ms on the 72-core Intel Xeon E5-2697 with the `-O3` compilation flag. This is only 3 times slower than the GPU performance leader — algorithm 4.16. Moreover, without compiler optimizations the execution time increases only slightly, as shown in Table 1.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

The observation that disabling OpenMP parallelization hardly changes the final execution time indicates that memory operations are the main limiting factor in this software implementation: the benefit of parallelization is limited here by intensive copying of data structures.

Nevertheless, on the 4-core CPU this specialized author-developed algorithm proved to be the best stable sorting variant among the considered ones.

4.6. The specialized author-developed TPT3 particle sorting algorithm implemented on the GPU using the CUDA Thrust library. The file `cuda_mysort_using_thrust.cu` in the repository [21] contains a GPU implementation of algorithm 4.5 using the CUDA Thrust library. The key points of the software implementation are shown in Listing 6. The distinguishing characteristics of this sorting variant remain the very small key-value range and the execution of the main sorting operations on the GPU.

Listing 6. The specialized TPT3 sorting algorithm implemented on GPU using CUDA Thrust

```
#include <cuda_runtime.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/transform.h>
#include <thrust/for_each.h>
...
for(int b=0; b<Nbin; ++b)
{
  thrust::sort(
                thrust::device,
                particles.begin()+init[b],
                particles.begin()+fin[b],
                [] __device__ __host__ (P & a, P & b)
                {
                  return a.ir<b.ir;
                }
  );
}
...
P* parray_pointer = thrust::raw_pointer_cast(particles.data());
int* count_minus1_pointer=thrust::raw_pointer_cast(count_minus1.data());
...
thrust::for_each(
  thrust::device,
  thrust::make_counting_iterator(0),
  thrust::make_counting_iterator(Nbin),
  [=] __device__ (int b)
  {
      for(int j=init_pointer[b]; j<fin_pointer[b]; ++j)
      {
        const P & pp=parray_pointer[j];
        if(-1 == pp.ir) count_minus1_pointer[b]++;
        else if(0  == pp.ir) count0_pointer[b]++;
        else if(1  == pp.ir) count1_pointer[b]++;
        else if(2  == pp.ir) count2_pointer[b]++;
        else if(3  == pp.ir) count3_pointer[b]++;
      }
  }
);
...
for(int b=0; b<Nbin; ++b)
{
  const int shift1=count_minus1_host_copy[b];
```

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

```
47    P* parray_dptr=thrust::raw_pointer_cast( particles.data() );
48    P* parray2_dptr=thrust::raw_pointer_cast( particles_out.data() );
49    cudaMemcpyAsync(
50            parray2_dptr+pointer_minus1_host_copy[b],
51            parray_dptr+init_host_copy[b],
52            count_minus1_host_copy[b] * sizeof(P),
53            cudaMemcpyDeviceToDevice);
54    ...
55  }
```

On the NVIDIA Tesla V100 GPU, the best result was achieved at $N_{\text{bin}} = 8192$ and was 41 ms. Within the measurement accuracy, no dependence on compiler optimizations was observed. This is the second best result among all considered GPU implementations. Thus, the computational scheme specialized for this problem is only slightly (about 1.5 times) slower on the GPU than the performance leader, namely the library radix sort for graphics accelerators.

4.7. Direct implementation of particle sorting on the GPU using the CUDA Thrust library. Listing 7 shows the simplest variant of a general-purpose GPU sorting implementation: a direct call to `thrust::sort` for the array of structures. This variant is important as a baseline library approach that does not require the development of a custom specialized sorting scheme.

Listing 7. Direct implementation of sorting on GPU using CUDA Thrust

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/sort.h>
4  ...
5  thrust::device_vector<P> particles_dev(N);
6  thrust::sort(thrust::device, particles_dev.begin(), particles_dev.end(),
7               [] __device__ __host__ (P & a, P & b){return a.ir>b.ir;});
8  thrust::host_vector<P> particles_check(N);
9  thrust::copy(particles_dev.begin(), particles_dev.end(), particles_out.begin());
```

On the NVIDIA Tesla V100 GPU, this variant gives an execution time in the range 91–93 ms at the `-O0` and `-O3` compilation flags. Although this result is about 4 times worse than the one obtained using algorithm 4.16, it is practically important: high performance is achieved by direct use of a library implementation, without the need to develop a custom specialized sorting scheme.

4.8. CPU particle sorting by splitting the original array into subarrays and applying `std::sort` to each of them in parallel using `std::thread`. In this variant, the original array is divided into N_{bin} subarrays, after which `std::sort` is launched independently for each subarray in a separate `std::thread` worker thread. Then particles with the same physical interaction type, already sorted within each subarray, are copied in parallel to the output data array with corresponding offsets. The complete software implementation of the algorithm is in the file `cpu_sorting_in_place.cpp` [21].

On the 4-core Intel Core i7-3770 CPU, the average execution time was 760 ms at `-O3` and 2 s at `-O0`. On the 64-core Intel Xeon Gold 6242 CPU, the corresponding times were 130 ms and 340 ms, and on the 72-core Intel Xeon E5-2697 they were 270 ms and 320 ms. Thus, on the many-core CPU, this approach is substantially faster than a call to `std::sort`.

If intermediate arrays are used to store the results of particle sorting in each subarray, instead of immediately copying the sorting results from each subarray to the final output array, the performance of the code decreases by 16% with `-O3` optimizations and by only 6% with `-O0`.

The number of threads, corresponding to N_{bin}, was set equal to the number of logical processor cores. It can be determined, for example, by the `$ nproc` command in Linux or by the C++ standard library function `std::thread::hardware_concurrency()`. However, numerical experiments showed that at $N_{\text{bin}} = 512$ substantially higher performance is achieved on CPUs. The corresponding measurement results are given in Table 1.

4.9. CPU particle sorting by splitting the original array into subarrays and applying `std::sort` to each of them in parallel using OpenMP. This algorithm is completely analogous to variant 4.8, but OpenMP is used for parallelization. Its source code is also contained in the file `cpu_sorting_in_place.cpp` [21].

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

On the 4-core Intel Core i7-3770 CPU, the average execution time was 700 ms at `-O3` and 1.5 s at `-O0`. On the 64-core Intel Xeon Gold 6242 CPU, the times were 120 ms and 250 ms, and on the 72-core Intel Xeon E5-2697 they were 220 ms and 270 ms. Among all numerical experiments aimed at determining computational performance, this variant showed the best result on the Intel Xeon Gold 6242 CPU.

The number of computational threads and the value of N_{bin} were initially chosen to be the same, also equal to the number of logical processor cores. However, after investigating the dependence of sorting performance on the number of subarrays, it was found that the maximum performance is achieved at $N_{\text{bin}} = 2048$: 92 ms on the 64-core Intel Xeon Gold 6242 CPU and 120 ms on the 72-core Intel Xeon E5-2697 at `-O3`, which is only 4–5 times slower than algorithm 4.16 on the GPU.

4.10. GPU particle sorting by splitting the original array into subarrays and applying `thrust::sort` to each of them in parallel. In algorithms 4.8 and 4.9, parallelization was performed on the CPU. Here the same general approach is implemented on the GPU using the CUDA Thrust library (file `cuda_sort_thrust_on_gpu.cu` [21]). On the NVIDIA Tesla V100 GPU, calculations for this sorting variant took 180 ms both at `-O3` and at `-O0`.

The CUDA Thrust library can launch code not only on NVIDIA graphics accelerators but also on CPUs. Therefore, the same code was recompiled for CPU execution and showed performance approximately 16, 2, and 3 times lower than on the NVIDIA Tesla V100 GPU on the Intel Core i7-3770, Xeon Gold 6242, and Xeon E5-2697 CPUs, respectively.

4.11. CPU particle sorting by splitting the original array into subarrays and applying `thrust::sort` to each of them in parallel. Because the CUDA Thrust library supports execution on both central processing units and graphics accelerators, sorting variant 4.10 was adapted for CPUs with minimal changes related to allocating memory on the CPU rather than on the GPU. The corresponding source code is given in the file `cuda_sort_thrust_on_cpu.cu` [21]. It was experimentally established that for the Intel Xeon Gold 6242 and Intel Xeon E5-2697 the best results are achieved at $N_{\text{bin}} = 4096$.

On the 4-core Intel Core i7-3770 CPU, the average execution time was 2.9 s at `-O3` and 45 s at `-O0`. On the 64-core Intel Xeon Gold 6242 CPU, the times were 410 ms and 3.3 s, and on the 72-core Intel Xeon E5-2697 they were 490 ms and 4.6 s. On the 4-core Intel Core i7-3770 CPU, the performance did not depend on N_{bin}, so on this computing platform the same value was used as on the others. This sorting variant is of interest as an experiment in moving the same code between CPUs and GPUs, but on CPUs it cannot compete with the best specialized solutions.

4.12. The particle sorting algorithm previously used in TPT3 on CPUs. Here we consider the particle sorting algorithm previously used in TPT3, which employed auxiliary intermediate arrays. It requires a large amount of additional memory and does not preserve the relative order of particles with the same interaction type. After the emergence of the faster stable solution 4.5, its usage was discontinued. Nevertheless, it was of interest to assess the potential performance of the former sorting algorithm when launched on the GPU.

This subsection considers the implementation of this algorithm for CPUs, given in the file `tpt3_old_sort_cpu.cpp` [21]. This sorting variant required 550 ms of execution time at `-O3` and 830 ms at `-O0` on the 4-core Intel Core i7-3770 CPU, 680 ms and 940 ms on the 64-core Intel Xeon Gold 6242 CPU and 590 ms and 920 ms on the 72-core Intel Xeon E5-2697, respectively. As can be seen, on the 4-core Intel Core i7-3770 CPU it is almost 4 times slower than the specialized TPT3 algorithm 4.5. At the same time, on the 64- and 72-core CPUs its performance is comparable with that of 4.5, whereas algorithms 4.8, 4.9 are several times faster. Moreover, as noted above, it requires allocation of a large amount of additional memory and does not preserve the relative order of transported particles with the same `ir`.

Thus, on CPUs this algorithm cannot be regarded as optimal in terms of either performance or allocated memory. Therefore, it is advisable to use instead algorithms 4.5, 4.8 and 4.9, which are substantially superior by these characteristics.

4.13. The particle sorting algorithm previously used in TPT3 on the GPU. This is the same algorithm as in the previous subsection 4.12, but ported to the GPU using the NVIDIA CUDA programming language for graphics accelerators. Its GPU implementation is in the file `cuda_sort_no_shared_memory.cu` [21], where a variant without CUDA shared memory is given.

On the NVIDIA Tesla V100 GPU, it showed an average execution time of about 220 ms independently of compiler optimization flags, which is only 9 times slower than the fastest GPU sorting variant — algorithm 4.16.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

4.14. The particle sorting algorithm previously used in TPT3 on the GPU using CUDA shared memory. This is the same algorithm as in 4.13, but using CUDA shared memory. Its program code is contained in the file `cuda_sort_shared_memory.cu` [21]. The strongly limited size of CUDA shared memory did not allow a direct run for the full particle array of size $N_\mathrm{p} = 2 \cdot 10^7$. Therefore, the task was run on smaller arrays, and the time was then extrapolated to the original size.

For this reason, the average execution time result for algorithm 4.14 should be interpreted with caution. The obtained estimates were 805 ms at `-O3` and 765 ms at `-O0`. These values are included in Table 1 only as a guideline and should not be regarded as fully comparable with direct measurements of the execution time of the other algorithms.

4.15. The particle sorting algorithm previously used in TPT3 on the GPU without CUDA shared memory, parallelized using CUDA streams. Here the sorting algorithm 4.13 was parallelized on the GPU using CUDA streams without using shared memory. Its software implementation is given in the file `cuda_threads_sort.cu` [21]. It was experimentally found that the best result on the NVIDIA Tesla V100 GPU is achieved with 16 CUDA streams.

The average execution time was 120 ms both at `-O3` and at `-O0`. Thus, even for the old algorithmic scheme, the transition to a graphics accelerator implementation gives a substantial performance increase. At the same time, using intermediate arrays to store the sorting results in each particle subarray, as was done at early stages of TPT3 development, not only doubles the amount of consumed memory but also increases the average execution time by 85% at `-O3` and by 43% at `-O0`.

4.16. The fastest particle sorting algorithm on the GPU using the NVIDIA CUDA CUB library. The fastest variant among the considered transported particle sorting methods turned out to be the `cub::DeviceRadixSort::SortPairs` call from the CUDA CUB library [18, 23]. This method belongs to library implementations of key-based radix sort and is well suited to the considered problem of sorting structures by the short integer field `ir`. In its implementation, a separate key array is formed, after which `SortPairs` permutes both the keys and the corresponding particle structures.

When using this library, an incompatibility problem arose between the versions of CUDA, the Thrust library, and the CUB library used. To eliminate the incompatibility between CUDA 11.4 and the CUB library, trial and error showed that the latest CUB release compatible with CUDA 11.4 is CUB 1.14. This release was used in the calculations. However, another problem then arose: CUB 1.14 was incompatible with the used Thrust 1.9.5 library, because it was much older. The only way to eliminate this error was to define the macro shown in Listing 8. This eliminated the compilation error caused by the absence of the corresponding header files and of the definition of this macro in the older CUDA CUB version. To compile the program using CUB, the `nvcc` compiler and the corresponding `-O0` and `O3` performance flags had to be used.

Listing 8. The macro needed to fix a compilation error caused by library incompatibility

```
1  #define THRUST_NS_QUALIFIER thrust
```

Listing 9 below shows an example of using `cub::DeviceRadixSort::SortPairs`. Here sorting was performed directly in the original array, without copying to an output array, and for clarity the particle array was given a name different from that in Listing 1.

Listing 9. The fastest particle sorting algorithm on GPU using the CUB library of NVIDIA CUDA

```
1  #include <cub/config.cuh>
2  #include <cub/util_type.cuh>
3  #include <cub/util_allocator.cuh>
4  #include <cub/util_namespace.cuh>
5  #include <cub/version.cuh>
6  #include <cub/device/device_radix_sort.cuh>
7  ...
8  {
9    P* h_points=new P[N];
10   int* h_keys=new int[N];
11   for(int i=0; i<N; i++)
12   {
13     h_points[i]={rand()%5-1,-9};
```

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

```
    h_keys[i]=h_points[i].ir;
  }
  P* d_points;
  int* d_keys;
  cudaMalloc(&d_points, N*sizeof(P));
  cudaMalloc(&d_keys, N*sizeof(int));
  cudaMemcpy(d_keys, h_keys, N*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_points, h_points, N*sizeof(P), cudaMemcpyHostToDevice);
  void* d_temp_storage=nullptr;
  size_t temp_storage_bytes;
  cub::DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes,
                                  d_keys, d_keys,
                                  d_points, d_points,
                                  N);
  cudaMalloc(&d_temp_storage, temp_storage_bytes);
  cub::DeviceRadixSort::SortPairs(d_temp_storage, temp_storage_bytes,
                                  d_keys, d_keys,
                                  d_points, d_points,
                                  N);
  ...
  delete [] h_points;
  delete [] h_keys;
  cudaFree(d_points);
  cudaFree(d_keys);
  cudaFree(d_temp_storage);
}
```

In accordance with the measurement methodology adopted in this work, the measured GPU execution time interval included only the *on-device* time of the sorting operation itself. For algorithm 4.16, at the beginning of the measurement the key array and the value array were already located on the GPU, and the required temporary buffer size had already been determined. Therefore, the measured interval characterizes specifically the time of the `cub::DeviceRadixSort::SortPairs` call for an already prepared *key/value* representation. Memory allocation, preparation of buffers on the host, host–device transfers, warm-up and auxiliary synchronization outside the measurement interval were not included in the measured execution time. It should be noted that the additional time delay associated with a one-time copying of the key array from the CPU to the GPU and back, in a scenario with repeated calls to a sorting procedure for data located on the GPU, is negligibly small and does not reduce performance when the number of sorting calls on the graphics accelerator is large. In the considered problem statement, this variant showed the best results: 27 ms at `-O0` and 24 ms at `-O3` on the NVIDIA Tesla V100 GPU.

From a practical standpoint, this means that in a scenario in which the data are already located in graphics accelerator memory and must be sorted repeatedly, the CUB library radix sort is the most preferable of all the considered GPU implementations.

5. Discussion of the results. The obtained results agree with theoretical expectations for sorting structures by a short integer key. Because sorting is performed by an integer key of cardinality $K = 5$, general-purpose comparison-based sorting algorithms are at a disadvantage relative to specialized schemes such as counting sort and radix sort. This is especially clear when `std::sort` from 4.1 is compared with algorithms 4.5, 4.6 and 4.16.

Among CPU library sorting variants, the strongest solution was `boost::sort::spreadsort::integer_sort` from 4.2. It is noticeably faster than `std::sort`, which is consistent with its hybrid nature and its better correspondence to the problem of sorting by an integer key. However, even this variant is inferior to the best specialized CPU sorting implementations in the considered problem statement.

On CPUs, the best result depends on the architecture and on the stability requirement. On the 4-core Intel Core i7-3770 CPU, the most successful variant was the specialized author-developed stable TPT3 algorithm 4.5, with an average execution time of 140 ms. On the 64-core Intel Xeon Gold 6242 CPU, the best times were shown by schemes with parallel subarray sorting 4.8 and 4.9 — about 90 ms. This means that when moving to many-core platforms software engineering aspects of the implementation become more important: parallelization, memory behavior and overheads associated with moving transported particle structures.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

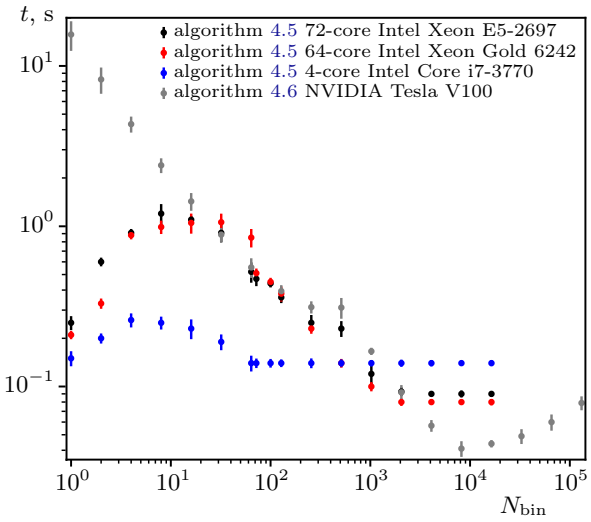

Fig. 1. The computation time for sorting an array of $2 \cdot 10^7$ particles with the help of algorithms 4.5, 4.6 depending on the value of subarray number N_{bin} on different computing platforms

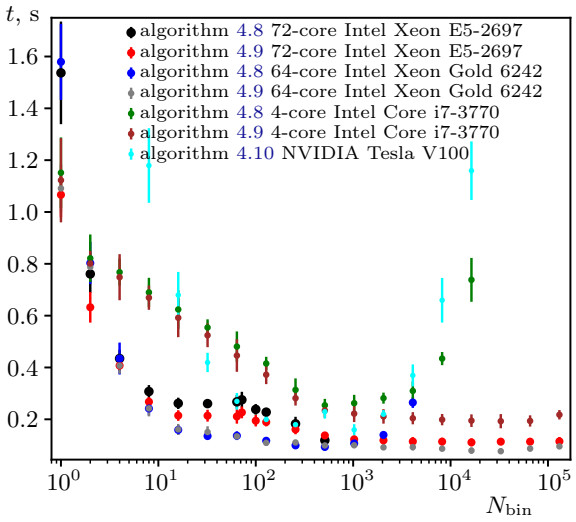

Fig. 2. The computation time for sorting an array of $2 \cdot 10^7$ particles with the help of algorithms 4.8–4.10 depending on the value of subarray number N_{bin} on different computing platforms

Algorithm 4.5 remains an important result precisely because it combines high performance with stability. Its performance on many-core CPUs is limited mainly by memory bandwidth. However, on a CPU with only 4 cores it showed one of the best results among stable CPU sorting variants. Therefore, the specialized scheme based on counting sort is valuable not only as a theoretically appropriate method for a small key range, but also as a practically efficient implementation.

On the GPU, the clear leader was `cub::DeviceRadixSort::SortPairs` 4.16: 24 ms at `-O3` on the NVIDIA Tesla V100. The closest sorting variant was the GPU implementation of the specialized author-developed TPT3 particle sorting algorithm using CUDA Thrust 4.6, with an execution time of 41 ms. A direct `thrust::sort` call 4.7, although inferior to the performance leader, nevertheless provides high computational speed with minimal programming effort.

It should be emphasized separately that the comparison of GPU code variants in this paper is performed only by *on-device* sorting time. For algorithm 4.16, this means that exactly the `SortPairs` stage is compared, with the *key/value* representation already prepared on the device. Within the adopted optimality criterion, such a comparison is correct and consistent with the general measurement methodology. This work did not aim to compare full *end-to-end* execution time including memory allocation and host–device data transfers. Therefore, the recommendation to use algorithm 4.16 applies primarily to problems similar to particle sorting in TPT3, where data remain in GPU memory for a long time and the sorting algorithm is invoked repeatedly.

The execution time result for algorithm 4.14 should not be interpreted on par with the direct performance measurements for the other algorithms, because it was obtained by extrapolation. By contrast, the comparison of algorithms 4.13 and 4.15 is useful from the standpoint of software engineering: it shows how strongly the quality of a GPU software implementation can affect the final computational performance even for the same basic algorithmic idea.

Of particular interest was the study of how the execution time of algorithms that use subarrays for sorting depends on the number of subarrays N_{bin}. For this purpose, the fastest algorithms among those using subarrays for parallel sorting were selected: 4.5, 4.8 and 4.9 on CPUs and 4.6 and 4.10 (as a full GPU analogue of 4.9) on graphics accelerators. The sorting time for an array of $N_{\text{p}} = 2 \cdot 10^7$ transported particles as a function of the number of subarrays N_{bin} is shown in Fig. 1 for algorithms 4.5 on the CPU and 4.6 on the GPU and in Fig. 2 for algorithms 4.8 and 4.9 on the CPU and 4.10 on the GPU.

Figure 1 shows that initially when N_{bin} does not exceed 100–1000 (depending on the considered CPU), the execution time increases. This occurs when the number of particles in the subarrays is large, i.e. exceeds tens of thousands for many-core CPUs and hundreds of thousands for the Intel Core i7-3770. Starting from $N_{\text{bin}} = 100$ for the 4-core CPU and from $N_{\text{bin}} = 1000$ for many-core CPUs, the execution time reaches a minimum and remains unchanged as the number of subarrays increases further. For many-core CPUs, this steady execution

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

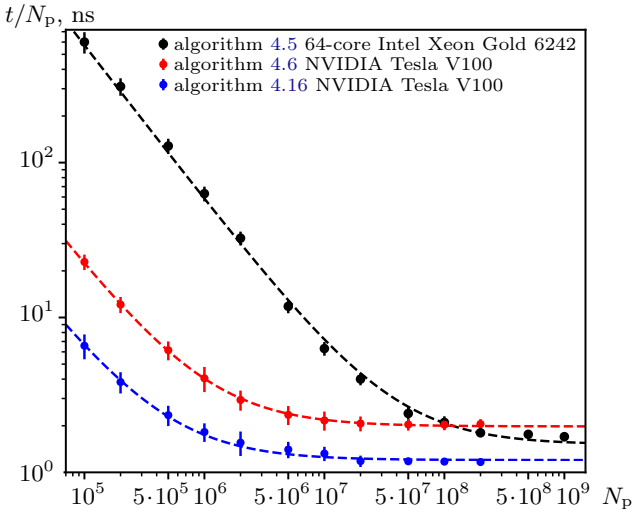

Fig. 3. The reduced computation time per primary particle of algorithms 4.5 on CPU and 4.6 and 4.16 on GPU at $N_\mathrm{bin} = 2048$

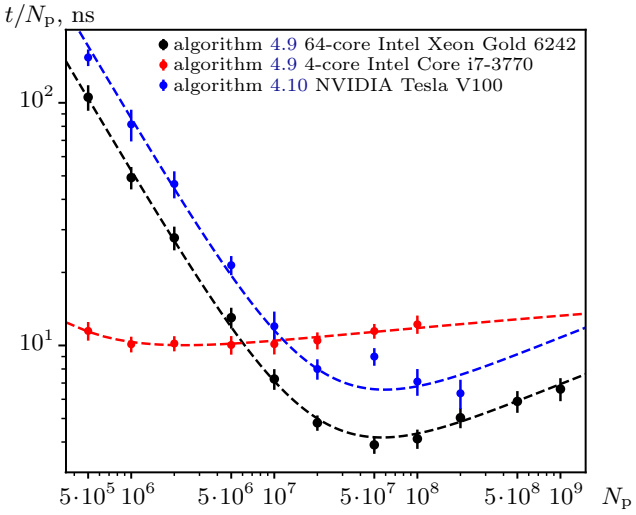

Fig. 4. The reduced computation time per primary particle of algorithms 4.9, 4.10 on different computing platforms at $N_\mathrm{bin} = 2048$

time is 2.5 times shorter than the execution time at $N_\mathrm{bin} = 1$, whereas for the 4-core Intel Core i7-3770 these times almost coincide. At the same time, the computation speed on the NVIDIA Tesla V100 GPU grows directly proportionally to the number of subarrays from $N_\mathrm{bin} = 1$ to approximately $N_\mathrm{bin} = 20$ and, starting from this value, it grows approximately proportionally to $\sqrt{N_\mathrm{bin}}$, reaching a maximum at $N_\mathrm{bin} = 8192$ (the use of this value for sorting algorithm 4.6 is reflected in Table 1). After that, a slight slowdown is observed.

The dependence of execution time on N_bin when using sorting algorithms 4.8 and 4.9 on different CPUs and 4.10 on the NVIDIA Tesla V100 GPU is shown in Fig. 2. Measurements of the execution time of algorithm 4.8 terminate at values of N_bin substantially smaller than for 4.9, because the used computing platforms had a system limit on the number of created computational threads that prevented moving to larger N_bin. Nevertheless, from the results for the 4-core CPU in Fig. 2, one can assume that the examined execution time of sorting algorithm 4.8 decreases as the number of computational threads (and, correspondingly, subarrays) grows, reaching a minimum at $N_\mathrm{bin} \sim 1000$ and then begins to increase again. At the same time, using OpenMP to parallelize sorting algorithm 4.9 leads to a continuous increase in execution speed as N_bin grows, reaching a maximum at $N_\mathrm{bin} \sim 2 \cdot 10^3$ and then remaining unchanged. Therefore, from Fig. 2, it can be stated with good accuracy that the maximum performance of algorithms 4.8, 4.9 on CPUs is observed at $N_\mathrm{bin} = 512$ and 2048, respectively, as reflected in Table 1. For algorithm 4.10 on the NVIDIA Tesla V100 GPU, the performance maximum when sorting $N_\mathrm{p} = 2 \cdot 10^7$ particles is reached at $N_\mathrm{bin} \sim 200$ and the dependence of execution time on the number of subarrays is even more pronounced, reaching 7.7 s at $N_\mathrm{bin} = 1$ and 2 s at $N_\mathrm{bin} = 32768$.

Figure 3 shows the reduced sorting time per particle as a function of N_p for algorithm 4.5 on the CPU, as well as for algorithms 4.6 and the fastest sorting algorithm 4.16 on the GPU. It was approximated in a unified way as $t/N_\mathrm{p} = t_1/N_\mathrm{p} + t_2$, where t is the total execution time, t_1 is the initialization time of the program code on the considered computing platform and t_2 is the reduced execution time per particle after the computation reaches an efficient intensive operating regime with high parallelization efficiency, when initialization costs become negligibly small. The reduced execution time of algorithm 4.5 on the 64-core Intel Xeon Gold 6242 CPU (black points) was described by the fitting function $5.71 \cdot 10^7/N_\mathrm{p} + 1.51$ ns, algorithm 4.6 on the GPU by $2.05 \cdot 10^6/N_\mathrm{p} + 1.98$ ns (red points) and algorithm 4.16 on the GPU by $5.46 \cdot 10^5/N_\mathrm{p} + 1.21$ ns (blue points). On the NVIDIA Tesla V100 GPU, the maximum particle-array size for which calculations were carried out was $N_\mathrm{p} = 2 \cdot 10^8$, because at this N_p the total memory allocated for the input and output particle arrays in computer memory is approximately 21 GB, whereas at $N_\mathrm{p} = 5 \cdot 10^8$ it already exceeds the graphics accelerator memory size of 32 GB. Thus, the largest initialization time is observed for sorting algorithm 4.5 on the 64-core Intel Xeon Gold 6242 CPU and is 57 ms. For algorithms 4.6 and 4.16 on the GPU, it is 2 μs and 0.5 μs, respectively. In terms of the reduced execution time per particle, algorithm 4.5 on a many-core CPU is 25% faster than 4.6 on the GPU and, in turn, is approximately the same amount slower than 4.16.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

Finally, Fig. 4 shows the dependence of the execution time of algorithms 4.9 on the CPU and 4.10 on the GPU on the number of particles N_p. In these algorithms, the standard `std::sort` sort, which has average asymptotic time complexity of $O(n \log n)$, where n is the number of elements being sorted, is used in parallel in each subarray. Therefore, by analogy with Fig. 3, the total execution time can be represented as $t = t_1 + t_2 \cdot N_\mathrm{p} \log N_\mathrm{p}$. In this regard, the reduced sorting time per particle for algorithm 4.9 on the 4-core Intel Core i7-3770 CPU was described by the function $1.5 \cdot 10^6/N_\mathrm{p} + 0.64 \cdot \log N_\mathrm{p}$ ns (red curve). The maximum number of particles on this CPU, for which calculations were performed, was $N_\mathrm{p} = 10^8$. At this value of N_p, the memory allocated for storing the input particle array is 5.2 GB, and for the input and output arrays it is already 10.4 GB, whereas the RAM of the 4-core Intel Core i7-3770 CPU is only 8 GB. Thus, in this case the computing device evidently remains operational due to paging and swap memory, whose size on this CPU is 11 GB. However, when N_p is doubled, even swap memory is insufficient and the computing device freezes, which explains the upper limit on N_p on the 4-core CPU. For the 64-core Intel Xeon Gold 6242 CPU and the NVIDIA Tesla V100 GPU, however, it was not possible to describe the execution time by this model dependence. The times were fitted instead by $5.17 \cdot 10^7/N_\mathrm{p} + 1.8 \cdot 10^{-6} \cdot \log^5 N_\mathrm{p}$ ns (algorithm 4.9, black curve) and $8.5 \cdot 10^7/N_\mathrm{p} + 2.8 \cdot 10^{-6} \cdot \log^5 N_\mathrm{p}$ ns (algorithm 4.10, blue curve), respectively. Therefore, it is not possible to interpret the physical meaning of the parameters in these dependences. At the same time, Fig. 4 shows that the minimum reduced sorting time per particle for algorithm 4.9 was 10 ns on the 4-core CPU and 4 ns on the 64-core CPU, while for sorting algorithm 4.10 on the NVIDIA Tesla V100 GPU it was 6.7 ns. This is several times larger than for the algorithms in Fig. 3.

Overall, the obtained data confirm that in the considered problem statement, there is no single "best" algorithm without qualifications. For CPUs, the choice depends on the architecture and the stability requirement. For GPUs, it depends on whether *on-device* or *end-to-end* time is compared. As noted above, however, when data remain in GPU memory for a long time and the sorting procedure is called repeatedly, the additional time delay associated with a one-time data copy from the CPU to the GPU and back is negligibly small, and the *on-device* execution time is decisive. Nevertheless, within the optimality criterion adopted in this paper, definite conclusions can be drawn for the specified data structure and key range.

6. Conclusion. This paper compared 16 implementations for sorting an array of $N_\mathrm{p} = 2 \cdot 10^7$ transported particles, described by the structure in Listing 1, by the integer field `ir`. All algorithms used the same input array with random `ir` values from the set $\{-1, 0, 1, 2, 3\}$.

It has been shown that for such a small key range, general-purpose comparison-based sorting algorithms are not the best choice in terms of performance. Among CPU implementations, the best results were shown by the specialized author-developed stable TPT3 sorting algorithm 4.5 on the 4-core Intel Core i7-3770 CPU (140 ms) and on many-core CPUs (80 and 90 ms on 64- and 72-core CPUs), as well as by schemes with parallel subarray sorting 4.8, 4.9 on the 64-core Intel Xeon Gold 6242 CPU (90 ms). Among CPU library sorting variants, `boost::sort::spreadsort::integer_sort` was the best in terms of performance, but it is inferior to the fastest algorithms.

On the NVIDIA Tesla V100 GPU, the fastest method was algorithm 4.16, based on radix sort from the CUDA CUB library: 24 ms at `-O3`. This value refers specifically to the sorting stage with data already prepared on the GPU. The closest variant in performance was algorithm 4.6, which is a GPU implementation of the specialized TPT3 sorting scheme 4.5 using the CUDA Thrust library, with an execution time of 41 ms.

Thus, within the optimality criterion adopted in this work, the most preferable option for the considered problem statement on the GPU is radix sort from the CUB library, whereas for CPUs the preferable options are specialized or specially parallelized software implementations adapted to the data structure and the small key range. These conclusions apply specifically to the considered transported particle structure and to the performance comparison by *on-device* GPU computation time. Transferring them to other data types and key ranges requires separate experimental verification.

References

1. R. M. Bergmann and J. L. Vujić, "Algorithmic Choices in WARP — A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs," Annals of Nuclear Energy **77**, 176–193 (2015). doi 10.1016/j.anucene.2014.10.039.
2. C. Liu, "Study on the Particle Sorting Performance for Reactor Monte Carlo Neutron Transport on Apple Unified Memory GPUs," EPJ Web of Conferences **302**, Article Number 04001 (2024). doi 10.1051/epjconf/202430204001.

2026, **27** (3), 317–333. doi 10.26089/NumMet.v27r321

3. J. Tramm, P. Romano, P. Shriwise, et al., "Performance Portable Monte Carlo Particle Transport on Intel, NVIDIA, and AMD GPUs," EPJ Web of Conferences **302**, Article Number 04010 (2024). doi 10.1051/epjconf/202430204010.
4. M. S. Egorova, S. A. Dyachkov, A. N. Parshikov, and V. V. Zhakhovsky, "Parallel SPH modeling using dynamic domain decomposition and load balancing displacement of Voronoi subdomains," Comp. Phys. Comm. **234**, 112–125 (2019). doi 10.1016/j.cpc.2018.07.019.
5. V. A. Vshivkov, T. V. Markelova, and V. I. Shelehov, "On Sorting Algorithms in the Particle-in-Cell Method," Nauch. Vestnik NGTU. **33** (4), 79–92 (2008). https://cyberleninka.ru/article/n/ob-algoritmah-sortirovki-v-metode-chastits-v-yacheykah. Cited June 7, 2026.
6. A. A. Romanenko and A. V. Snytnikov, "Optimization of Model Reordering Optimization for GPU Implementation of Particle-in-Cell Method," Vestnik NGU. Ser.: Informat. Technol. **17** (1), 82–89 (2019). doi 10.25205/1818-7900-2019-17-1-82-89.
7. N. V. Snytkov, O. P. Stoyanovskaya, and T. A. Glushko, "Parallel Algorithm for Supercomputing Simulation of Dust-Gaseous Gravitating Systems Using Particle-in-Cell and SPH Methods," Journal of Physics: Conference Series **1336** (1), Article Number 012021 (2019). doi 10.1088/1742-6596/1336/1/012021.
8. S. Agostinelli, J. Allison, K. Amako, et al., "Geant4 — A simulation toolkit," Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **506** (3), 250–303 (2003). doi 10.1016/S0168-9002(03)01368-8.
9. J. Allison, K. Amako, J. Apostolakis, et al., "Geant4 Developments and Applications," IEEE Transactions on Nuclear Science **53** (1), 270–278 (2006). doi 10.1109/TNS.2006.869826.
10. A. A. Galyuzov and M. V. Kosov, "Increasing Performance of the Radiation Transport Simulation: TPT3 Parallel Program," Software & Systems **38** (2), 269–279 (2025). doi 10.15827/0236-235X.150.269-279.
11. A. A. Galyuzov and M. V. Kosov, *The TPT3 program for high-performance simulation of radiation transfer*, Certificate of RF Registration of Computer Program №. 2024614877. Date of Registration: 29.02.2024.
12. OpenMP Preprocessor Directive Standard. https://www.openmp.org.
13. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda/toolkit. Cited June 7, 2026.
14. OpenACC Preprocessor Directive Standard. https://www.openacc.org. Cited June 7, 2026.
15. D. E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching. 2nd ed.* (Addison-Wesley, Boston, 1998).
16. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms. 3rd ed.* (MIT Press, Cambridge, 2009).
17. S. J. Ross, "The Spreadsort High-Performance General-Case Sorting Algorithm," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications **3**, 1100–1106 (2002).
18. N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *2009 IEEE International Symposium on Parallel & Distributed Processing, Italy, Rome, May 23–29, 2009* (IEEE Press, 2009), pp. 1–10. doi 10.1109/IPDPS.2009.5161005.
19. D. R. Musser, "Introspective Sorting and Selection Algorithms," Software: Practice and Experience **27** (8), 983–993 (1997).
20. A. A. Galyuzov and M. V. Kosov, "Calculation of the Gamma Quanta Yield for the $^{(235)}$U Fission in the Uranium Cube by the TPT3 Program," Technol. EMC. **89** (2), 26–32 (2024).
21. The repository with the source code for the examples from the article. https://github.com/Agar92/benchmarksortingalgorithms.git. Cited June 7, 2026.
22. Boost.Sort Documentation. https://www.boost.org/libs/sort/. Cited June 7, 2026.
23. The repository with the source code for the CUDA CUB library. https://github.com/NVIDIA/cub.git. Cited June 7, 2026.

Received	*Accepted*	*Published*
April 30, 2026	*May 24, 2026*	*July 1, 2026*

Information about the author

Andrey A. Galyuzov — Senior Researcher; All-Russian Research Institute of Automation named after N. L. Dukhov (VNIIA), Suschevskaya ulitsa, 22, 127030, Moscow, Russia.