2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

doi 10.26089/NumMet.v27r321 УДК 004.424

# 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. 317–336. doi 10.26089/NumMet.v27r321.

2026, **27** (3), 317–336. 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), 317–336 (2026). doi 10.26089/NumMet.v27r321.

---

**1. Введение.** Задачи сортировки больших массивов малоразмерных структур данных по короткому целочисленному ключу регулярно возникают в программных комплексах моделирования переноса частиц, когда требуется сгруппировать объекты по типу физического взаимодействия, состоянию или другой дискретной характеристике [1–3]. Помимо этого, сортировка моделируемых частиц, находящихся в одной и той же ячейке расчетной области, по различным параметрам с целью уменьшения числа кэш-промахов также используется, например в программах молекулярной динамики, PIC-моделирования и гидродинамики [4–7]. От выбора способа сортировки в таких задачах зависит не только время самой процедуры упорядочивания, но и локальность данных, а следовательно, эффективность последующих этапов вычислений.

В универсальном программном комплексе Geant4 [8, 9] и в специализированных кодах переноса радиации задача группировки частиц по типу взаимодействия возникает естественным образом. В программе высокопроизводительного параллельного Монте-Карло моделирования радиационного переноса TPT3 (Toolkit for Particle Transport 3, фреймворк для моделирования транспорта частиц версии 3) [10, 11], работающей на CPU при использовании OpenMP [12] и на GPU при использовании CUDA [13] и OpenACC [14], такая перегруппировка используется как часть вычислительного конвейера и непосредственно влияет на производительность параллельных вычислений. Она нужна для увеличения локальности однотипных данных, необходимой для их эффективной параллельной обработки и уменьшения числа кэш-промахов.

Важно подчеркнуть, что вопрос о "лучшем" алгоритме сортировки имеет смысл только после уточнения модели задачи. Для сортировок, основанных на сравнениях, действует нижняя оценка $O(n \log n)$ по

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

числу операций сравнения [15, 16]. В то же время для целочисленных ключей ограниченного диапазона возможны линейные по $n$ методы, основанные на подсчете, поразрядной обработке или распределении по корзинам [16–18]. Следовательно, универсального алгоритма, одинаково оптимального для всех постановок задачи, не существует: выбор определяется диапазоном ключей, размером структуры переносимой частицы, требованиями к устойчивости, доступной памятью и конкретной вычислительной платформой.

В библиотечных реализациях общего назначения это различие хорошо видно. На CPU в качестве базового алгоритма часто используются гибридные схемы семейства *introsort* [19], а для целочисленных ключей применяются специализированные гибридные методы, например *spreadsort* [17]. На GPU широкое распространение получили варианты поразрядной сортировки и сортировки по ключу, ориентированные на высокую пропускную способность памяти [18].

В программе TPT3 сортируется массив структур переносимых частиц по целочисленному полю `ir`, обозначающему тип физического взаимодействия. В текущей постановке задачи поле `ir` принимает всего пять значений: от $-1$ до 3. Поэтому сравнение универсальных алгоритмов сортировки, основанных на операциях сравнения, и специализированных методов для целочисленного ключа здесь особенно показательно.

В данной работе под *оптимальным* далее понимается алгоритм с наименьшим средним временем выполнения при фиксированных входных данных на конкретной вычислительной платформе с общей оперативной памятью. Если средние значения времени близки, преимуществом считается меньший объем дополнительной памяти. Устойчивость сортировки, т.е. сохранение относительного порядка элементов с одинаковым значением ключа, рассматривается как важная, но не первичная характеристика.

Цель работы — сравнить 16 реализаций сортировки массива частиц в программе TPT3 на CPU и GPU, определить наиболее производительные варианты для рассматриваемой структуры частицы и указать ограничения применимости полученных выводов. Работа не претендует на поиск универсально лучшего алгоритма сортировки. Анализ относится к сортировке массива структур вида, приведенного в листинге 1, по маломощному целочисленному ключу `ir` на конкретной вычислительной платформе с общей оперативной памятью.

**2. Постановка задачи и критерии сравнения.** В тестовой задаче сортируется массив из $N_\mathrm{p} = 2 \cdot 10^7$ структур переносимых частиц, что требует выделения примерно 1 ГБ памяти вычислительного устройства. Размер входного массива выбран максимальным, так, чтобы объем потребляемой памяти да-

Листинг 1. Минимальное описание структуры переносимой частицы
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), 317–336. doi 10.26089/NumMet.v27r321

же самых ресурсоемких алгоритмов умещался в RAM объемом 8 ГБ компьютера с CPU Intel Core i7-3770, использовавшегося для расчетов. Описание структуры частицы дано в листинге 1, после которого приводятся объявления входного и выходного массивов частиц. Далее используется нейтральное обозначение "структура" (а не POD-структура), поскольку в указанном типе есть пользовательские конструкторы и перегруженные операторы.

Целочисленное поле `ir` служит ключом сортировки. В настоящей версии программы ТРТ3 оно принимает значения из множества $\{-1, 0, 1, 2, 3\}$, т.е. число различных ключей $K$ равно 5. При таком малом диапазоне ключа следует ожидать преимущества методов сортировки подсчетом/поразрядной сортировки над универсальными алгоритмами, основанными на сравнениях.

Для всех рассматриваемых реализаций использовался один и тот же входной массив из $N_\mathrm{p}$ структур. Значения поля `ir` во входном массиве генерировались случайно, равномерно по указанному диапазону. Остальные поля структуры переносились вместе с ней как связанные данные. Перед каждым запуском сортировки на вход подавалась одна и та же заранее подготовленная копия этого массива, что позволило напрямую сравнивать среднее время выполнения разных алгоритмов.

Хотя устойчивость сортировки не была главным критерием выбора алгоритма, эта характеристика в ряде прикладных и отладочных сценариев оказывается полезной. Например, при анализе и верификации расчетов, описанных в [20], сохранение относительного порядка нейтронов с одинаковым типом взаимодействия существенно упрощало поиск ошибок в программной реализации алгоритма весового моделирования цепной реакции ядерного деления.

**3. Методика измерений.** Было рассмотрено 16 реализаций сортировки. Сравнивались среднее время выполнения, устойчивость сортировки и требование к дополнительной памяти. Основным критерием оптимальности служило среднее время выполнения. Минимальный объем вспомогательной памяти использовался как дополнительный критерий при близких средних значениях времени счета.

Каждая вычислительная задача запускалась $m = 3, 4, 5$ раз после предварительного "прогрева" (warm-up) вычислительного устройства, т.е. тестового исполнения на нем несколько раз этой же задачи с целью выхода его процессора и кэш-памяти на оптимальный интенсивный режим работы. По значениям времени отдельных запусков $t_1, \ldots, t_m$ вычислялись выборочное среднее

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

и выборочное стандартное отклонение

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

Полуширина двустороннего 95%-го доверительного интервала для среднего времени выполнения программного кода определялась по формуле

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

где $t_{0.975,m-1}$ — квантиль распределения Стьюдента с $m-1$ степенями свободы. В табл. 1 приводится среднее время $\bar{t}$, округленное с учетом величины $\Delta t$, чтобы не перегружать представление результатов полным набором статистических данных.

На GPU измерялось только *on-device* время сортировки. Оно определялось при помощи CUDA events, установленных непосредственно перед вызовом сортировки и сразу после него. Поэтому в измерение времени счета не включались выделение памяти, переносы данных между CPU (host) и GPU (device), warm-up, служебная синхронизация GPU вне измеряемого интервала и прочие накладные расходы среды выполнения. Следовательно, результаты для GPU следует интерпретировать как результаты для сценария, в котором данные уже расположены в памяти устройства.

В табл. 1 в первом столбце стоит номер алгоритма, совпадающий с порядковым номером подраздела в разделе 4 данной работы. Во втором столбце отмечается, обеспечивается ли сохранение относительного порядка частиц с одинаковым значением `ir`. Для устойчивых алгоритмов это достигается свойством самой сортировки. Для части алгоритмов, основанных на сравнениях, тот же эффект обеспечивался дополнительным упорядочиванием по полю `id`. В третьем столбце указано число подмассивов $N_\mathrm{bin}$, а в столбцах

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

Таблица 1. Таблица сравнительной производительности различных программных алгоритмов сортировки массива из $2 \cdot 10^7$ структур переносимых частиц на различных вычислительных платформах. Номер алгоритма в первом столбце совпадает с порядковым номером подраздела в разделе 4 данной работы

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

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

* Среднее время для алгоритма 4.14 получено экстраполяцией по измерениям на уменьшенных массивах; поэтому этот результат следует рассматривать как ориентировочный

$R$(`-O0`) и $R$(`-O3`) приводится отношение среднего времени счета соответствующего варианта алгоритма сортировки к среднему времени работы алгоритма 4.16 при той же степени компиляторной оптимизации. Эта относительная шкала используется только как компактный способ представления результатов внутри рассматриваемого набора численных экспериментов и не предназначена для прямого архитектурного сопоставления CPU и GPU.

Репозиторий GitHub с исходным кодом примеров из статьи расположен по адресу [21]. Указанные далее в тексте статьи файлы находятся в нем.

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

Таблица 2. Сравнительные характеристики использовавшихся при расчетах вычислительных платформ
Table 2. Comparative characteristics of used computing platforms

| Платформа | Число ядер<br>Number of cores | Число потоков<br>Number of threads | Базовая частота, ГГц<br>Basic frequency, GHz | Техпроцесс, нм<br>Technology, nm | Размер кристалла, мм$^2$<br>Chip size, mm$^2$ | Транзисторы, млрд<br>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 ядер CUDA<br>5120 CUDA cores | 5120 | 1.4 | 12 | 815 | 21.1 |

В табл. 2 приведены основные характеристики использованных вычислительных платформ. Для CPU Intel Xeon Gold 6242 и Intel Xeon E5-2697 в таблице указаны характеристики одного процессора, тогда как в экспериментах использовались вычислительные узлы в двухсокетной компоновке. Поэтому далее обозначения “64-ядерный Intel Xeon Gold 6242” и “72-ядерный Intel Xeon E5-2697” относятся к полному вычислительному узлу и означают соответственно 64 и 72 логических ядра двух процессоров. Для GPU значения в столбцах “Число ядер” и “Число потоков” приведены в терминологии спецификации устройства и не используются как самостоятельная основа для количественного сопоставления с CPU. В работе сравниваются именно зафиксированные аппаратные конфигурации. Прямое сопоставление по числу транзисторов или площади кристалла не использовалось как самостоятельный критерий объяснения сравнительной производительности.

## 4. Результаты численных экспериментов.

**4.1. Стандартная сортировка C++ `std::sort`.** В данной работе `std::sort` используется как основанный на сравнениях базовый вариант алгоритма сортировки общего назначения (листинг 2). В типичных реализациях стандартной библиотеки он относится к гибридным алгоритмам семейства `introsort` [19], однако стандарт C++ не фиксирует конкретную внутреннюю реализацию, поэтому здесь нас интересует прежде всего быстродействие на выбранных платформах.

Для рассматриваемой задачи `std::sort` уступает методам, использующим малый диапазон ключа `ir`. На 4-ядерном CPU Intel Core i7-3770 среднее время выполнения без сохранения порядка следования частиц с одинаковым `ir` составляет 1.04 с при `-O3` и 5.7 с при `-O0`; при добавлении требования сохранения порядка — соответственно 2.1 с и 7.8 с. На 64-ядерном Intel Xeon Gold 6242 без сохранения порядка получено 980 мс при `-O3` и 5.3 с при `-O0`, а на 72-ядерном Intel Xeon E5-2697 — 950 мс и 5.3 с соответственно.

Сортировка без сохранения порядка определяется компаратором `l.ir > r.ir`. Для варианта, в котором требовалось воспроизводимое для определенных задач упорядочение частиц с одинаковым `ir` по возрастанию поля `id`, в компаратор добавлялось вторичное упорядочивание по полю `id`:

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

Такой режим работы алгоритма сортировки при предположении, что поле `id` монотонно соответствует росту возраста частицы, обеспечивает сохранение требуемого относительного порядка элементов внутри групп частиц с одинаковым `ir`.

Чтобы не перегружать дальнейшее изложение, в последующих подразделах для неустойчивых алгоритмов отдельно обсуждается только их вариант без сохранения относительного порядка следования частиц с одинаковым типом взаимодействия.

Листинг 2. Использование алгоритма стандартной сортировки C++ `std::sort`
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;});
```

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

Листинг 3. Использование алгоритма сортировки по целочисленным значениям `integer_sort` из комбинированного высокопроизводительного сортировочного алгоритма `spreadsort` библиотеки Boost Sort

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

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

**4.2. Сортировка из библиотеки Boost Sort.** Путем численных экспериментов было установлено, что из алгоритмов библиотеки Boost Sort для рассматриваемой задачи наилучший результат дает семейство `boost::sort::spreadsort`. Этот алгоритм сочетает идеи поразрядной и основанной на сравнениях сортировки [17], а его библиотечная реализация описана в [22]. Поскольку сортировка выполняется по целочисленному полю, применялся вариант `boost::sort::spreadsort::integer_sort`, использование которого приводится в листинге 3.

На 4-ядерном CPU Intel Core i7-3770 и на 64-ядерном Intel Xeon Gold 6242 средние значения времени выполнения составили 270 мс при `-O3` и, соответственно, 1.1 с и 1 с при `-O0`. На 72-ядерном Intel Xeon E5-2697 получено около 340 мс при `-O3` и 1.2 с при `-O0`. Тем самым `spreadsort` заметно быстрее стандартной сортировки, основанной на сравнениях, и служит сильным библиотечным базовым вариантом для CPU, однако в рассматриваемой постановке задачи все же уступает более специализированным схемам.

**4.3. Стандартная сортировка и GPU offloading при помощи execution policy при использовании компилятора NVIDIA HPC SDK.** Этот вариант интересен как наименее трудоемкий способ перенести сортировку на GPU, практически не меняя исходный код на C++. В листинге 4 показано использование политики выполнения `std::execution::par` и флага `-stdpar=gpu` компилятора `nvc++`. В сравнении участвовало только время *on-device*, поэтому данный результат следует интерпретировать именно как оценку эффективности библиотечной реализации выгрузки кода на графический ускоритель в выбранной среде.

На узле сервера с 64-ядерным CPU Intel Xeon Gold 6242 и GPU NVIDIA Tesla V100 такой вариант считался 340 мс. В пределах точности измерений зависимость от флагов компиляции `-O0` или `-O3` не наблюдалась. Этот способ удобен тем, что не требует явной реализации на CUDA, однако по производительности он уступает специализированным методам для GPU.

**4.4. Стандартный алгоритм поразрядной сортировки.** В файле `radixsort.cpp` репозитория [21] приведена реализация стандартной поразрядной сортировки на CPU. Теоретически поразрядная сортировка хорошо согласуется с задачами сортировки по целочисленному ключу [16, 18], однако практический результат определяется деталями реализации, характером перемещаемых данных и организацией памяти.

В рассматриваемой реализации поразрядная сортировка полного массива структур оказалась существенно медленнее более удачных по производительности вариантов. На 4-ядерном CPU Intel Core i7-3770 среднее время выполнения составляет 2 с при `-O3` и 2.1 с при `-O0`. На 64-ядерном Intel Xeon Gold 6242 получено 2.5 с и 2.7 с, а на 72-ядерном Intel Xeon E5-2697 — 2.3 с и 2.5 с соответственно. Следовательно,

Листинг 4. Использование стандартной сортировки и GPU offloading при помощи execution policy с компилятором NVIDIA HPC SDK

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());
```

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

сама принадлежность алгоритма к классу поразрядной сортировки еще не гарантирует высокой производительности; решающую роль играет конкретная реализация.

**4.5. Специализированный авторский алгоритм сортировки частиц в TPT3.** В листинге 5 и файле `benchmark.cpp` репозитория [21] приведен специализированный алгоритм, разработанный для программы TPT3. По сути это усложненный и модернизированный метод сортировки подсчетом. Сначала исходный массив разбивается на $N_{\text{bin}}$ подмассивов, затем для каждого подмассива подсчитывается число частиц с каждым значением `ir`, далее вычисляются префиксные смещения, после чего элементы копируются в выходной массив в нужные диапазоны.

Листинг 5. Специализированный авторский алгоритм сортировки структур частиц в TPT3

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++)
  {
    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
```

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

```
  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];
    }
  }
}
```

При фиксированном числе различных ключей $K = 5$ вычислительная сложность такого подхода составляет $O(N_\mathrm{p})$, а дополнительная память — $O(N_\mathrm{p} + K\,N_\mathrm{bin})$ из-за наличия выходного массива и вспомогательных счетчиков. Алгоритм сохраняет относительный порядок частиц с одинаковым значением `ir`, поскольку внутри каждого подмассива копирование в выходной массив выполняется в исходном порядке. В этом смысле он является специализированным устойчивым вариантом идеи сортировки подсчетом [16].

На 4-ядерном CPU Intel Core i7-3770 среднее время выполнения составляет около 140 мс при `-O3` и 150 мс при `-O0` при выборе $N_\mathrm{bin} = 4$. На 64-ядерном CPU Intel Xeon Gold 6242 при $N_\mathrm{bin} = 64$ получено 530 мс при `-O3` и около `650 мс` при `-O0`, а на 72-ядерном центральном процессоре Intel Xeon E5-2697 при $N_\mathrm{bin} = 72$ — порядка 470–520 мс.

Число подмассивов $N_\mathrm{bin}$ стандартно выбиралось равным числу логических ядер соответствующего центрального процессора. Однако было обнаружено, что при $N_\mathrm{bin}$=2048 на многоядерных центральных процессорах удается достичь существенно большего быстродействия: 80 мс на 64-ядерном CPU Intel Xeon Gold 6242 и 90 мс на 72-ядерном Intel Xeon E5-2697 с флагом компиляции `-O3`, что всего в 3 раза уступает по скорости счета лидеру производительности на GPU — алгоритму 4.16, причем без оптимизаций компилятора время счета почти не увеличивается, что показано в табл. 1.

Наблюдение, что отключение распараллеливания при помощи OpenMP практически не меняет итоговое время счета, указывает на то, что операции с памятью являются основным ограничивающим фактором данной программной реализации: выигрыш от распараллеливания здесь ограничивается интенсивным копированием структур данных.

Тем не менее, на 4-ядерном CPU этот специализированный авторский алгоритм оказался лучшим устойчивым вариантом сортировки из рассмотренных.

**4.6. Специализированный авторский алгоритм сортировки частиц в TPT3, реализованный на GPU при помощи библиотеки CUDA Thrust.** В файле `cuda_mysort_using_thrust.cu` репозитория [21] приведена реализация алгоритма 4.5 на GPU с использованием библиотеки CUDA Thrust. Ключевые моменты программной реализации представлены в листинге 6. Отличительными характеристиками этого варианта сортировки остаются неизменно малый диапазон значений ключа и выполнение основных операций сортировки на GPU.

Листинг 6. Специализированный авторский алгоритм сортировки в TPT3, реализованный на GPU при помощи библиотеки CUDA Thrust

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>
```

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

```
...
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];
  P* parray_dptr=thrust::raw_pointer_cast( particles.data() );
  P* parray2_dptr=thrust::raw_pointer_cast( particles_out.data() );
  cudaMemcpyAsync(
          parray2_dptr+pointer_minus1_host_copy[b],
          parray_dptr+init_host_copy[b],
          count_minus1_host_copy[b] * sizeof(P),
          cudaMemcpyDeviceToDevice);
  ...
}
```

На GPU NVIDIA Tesla V100 наилучший результат достигался при $N_{\text{bin}} = 8192$ и составил 41 мс; в пределах точности измерений зависимости от компиляторных оптимизаций не наблюдалось. Это второй результат среди всех рассмотренных реализаций на GPU. Таким образом, специализированная для данной задачи расчетная схема на GPU незначительно (примерно в 1.5 раза) проигрывает в производительности лидеру в скорости счета — библиотечной поразрядной сортировке на графических ускорителях.

**4.7. Непосредственная реализация сортировки частиц на GPU при помощи библиотеки CUDA Thrust.** Листинг 7 показывает простейший вариант реализации сортировки общего назначения на GPU: прямой вызов *thrust::sort* для массива структур. Этот вариант важен как базовый библиотечный подход, не требующий разработки собственной специализированной схемы сортировки.

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

Листинг 7. Непосредственная реализация сортировки на GPU при помощи CUDA Thrust
Listing 7. Direct implementation of sorting on GPU using CUDA Thrust

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

На GPU NVIDIA Tesla V100 он обеспечивает скорость счета в диапазоне 91–93 мс при флагах компиляции `-O0` и `-O3`. Хотя этот результат примерно в 4 раза хуже полученного с помощью алгоритма 4.16, он имеет важное практическое значение: высокое быстродействие достигается за счет прямого использования библиотечной реализации, без необходимости в разработке собственной специализированной схемы сортировки.

**4.8. Сортировка частиц на CPU путем разбиения исходного массива на подмассивы и использования параллельно (при помощи `std::thread`) для каждого из них `std::sort`.** В этом варианте исходный массив разбивается на $N_{\text{bin}}$ подмассивов, после чего для каждого из них независимо запускается `std::sort` в отдельном рабочем потоке `std::thread`. Затем отсортированные в каждом подмассиве частицы с одинаковым типом физического взаимодействия параллельно с соответствующими отступами копируются в массив выходных данных. Полная программная реализация алгоритма находится в файле `cpu_sorting_in_place.cpp` [21].

На 4-ядерном CPU Intel Core i7-3770 среднее время выполнения составило 760 мс при `-O3` и 2 с при `-O0`. На 64-ядерном CPU Intel Xeon Gold 6242 получено 130 мс и 340 мс, а на 72-ядерном Intel Xeon E5-2697 — 270 мс и 320 мс. Таким образом, на многоядерном CPU этот подход оказывается существенно быстрее вызова `std::sort`.

Если использовать промежуточные массивы для хранения результатов сортировки частиц в каждом подмассиве, а не сразу копировать результаты сортировки частиц в каждом подмассиве в конечный выходной массив, то производительность кода при оптимизациях с `-O3` уменьшается на 16%, а с `-O0` — всего на 6%.

Число потоков, соответствующее $N_{\text{bin}}$, задавалось равным числу логических ядер процессора. Его можно определить, например, командой `$ nproc` в Linux или функцией стандартной библиотеки C++ `std::thread::hardware_concurrency()`. Однако путем численных экспериментов было найдено, что при $N_{\text{bin}}$=512 на центральных процессорах достигается существенно большая производительность, результаты измерения которой приводятся в табл. 1.

**4.9. Сортировка частиц на CPU путем разбиения исходного массива на подмассивы и использования параллельно (при помощи OpenMP) для каждого из них `std::sort`.** Этот алгоритм полностью аналогичен варианту 4.8, но для распараллеливания используется OpenMP. Его исходный код также находится в файле `cpu_sorting_in_place.cpp` [21].

На 4-ядерном CPU Intel Core i7-3770 среднее время выполнения составило 700 мс при `-O3` и 1.5 с при `-O0`. На 64-ядерном центральном процессоре Intel Xeon Gold 6242 получено 120 мс и 250 мс, а на 72-ядерном Intel Xeon E5-2697 — 220 мс и 270 мс. Из всех численных экспериментов по определению производительности вычислений именно этот вариант показал лучший результат на CPU Intel Xeon Gold 6242.

Число вычислительных потоков и значение $N_{\text{bin}}$ изначально выбирались одинаковыми, также равными количеству логических ядер процессора. Но после исследования зависимости производительности сортировки от числа подмассивов было найдено, что ее максимальная производительность достигается при $N_{\text{bin}}$=2048: 92 мс на 64-ядерном CPU Intel Xeon Gold 6242 и 120 мс на 72-ядерном Intel Xeon E5-2697 при `-O3`, что всего в 4–5 раз медленнее, чем 4.16 на GPU.

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

**4.10. Сортировка частиц на GPU путем разбиения исходного массива на подмассивы и использования параллельно для каждого из них `thrust::sort`.** В алгоритмах 4.8 и 4.9 распараллеливание выполнялось на CPU. Здесь тот же общий подход реализован на GPU при помощи библиотеки CUDA Thrust (файл `cuda_sort_thrust_on_gpu.cu` [21]). На GPU NVIDIA Tesla V100 расчеты такого варианта сортировки длились 180 мс как при `-O3`, так и при `-O0`.

Библиотека CUDA Thrust позволяет запускать код не только на графических ускорителях NVIDIA, но и на центральных процессорах. Поэтому тот же самый код был перекомпилирован для запуска на CPU и показал примерно в 16, 2 и 3 раза меньшую, чем на графическом ускорителе NVIDIA Tesla V100, производительность на центральных процессорах Intel Core i7-3770, Xeon Gold 6242 и Xeon E5-2697.

**4.11. Сортировка частиц на CPU путем разбиения исходного массива на подмассивы и использования параллельно для каждого из них `thrust::sort`.** Поскольку библиотека CUDA Thrust поддерживает запуск как на центральных процессорах, так и на графических ускорителях, вариант сортировки 4.10 был с минимальными изменениями, связанными с выделением памяти на CPU, а не на GPU, адаптирован для центральных процессоров. Соответствующий исходный код приведен в файле `cuda_sort_thrust_on_cpu.cu` [21]. Экспериментально было установлено, что для Intel Xeon Gold 6242 и Intel Xeon E5-2697 наилучшие результаты достигаются при $N_{\text{bin}} = 4096$.

На 4-ядерном CPU Intel Core i7-3770 среднее время выполнения составило 2.9 с при `-O3` и 45 с при `-O0`. На 64-ядерном центральном процессоре Intel Xeon Gold 6242 получено 410 мс и 3.3 с, а на 72-ядерном Intel Xeon E5-2697 — 490 мс и 4.6 с. На 4-ядерном центральном процессоре Intel Core i7-3770 быстродействие не зависело от величины $N_{\text{bin}}$, поэтому на данной вычислительной платформе эта величина бралась той же, что и на остальных. Этот вариант сортировки интересен как эксперимент с переносом одного и того же кода между CPU и GPU, но на центральных процессорах он не в состоянии конкурировать с лучшими специализированными решениями.

**4.12. Ранее использовавшийся в TPT3 алгоритм сортировки частиц на CPU.** Здесь рассматривается ранее использовавшийся в программе TPT3 алгоритм сортировки частиц при помощи вспомогательных промежуточных массивов. Он требует большого объема дополнительной памяти и не сохраняет относительный порядок следования частиц с одинаковым типом взаимодействия. После появления более быстрого устойчивого решения 4.5 его использование было прекращено. Но представляла интерес потенциальная производительность прежнего алгоритма сортировки, если запустить его на GPU.

Здесь рассматривается реализация этого алгоритма для запуска на центральных процессорах, которая приводится в файле `tpt3_old_sort_cpu.cpp` [21]. Этот вариант сортировки считался 550 мс при `-O3` и 830 мс при `-O0` на 4-ядерном CPU Intel Core i7-3770, соответственно, 680 мс и 940 мс на 64-ядерном центральном процессоре Intel Xeon Gold 6242 и 590 мс и 920 мс на 72-ядерном Intel Xeon E5-2697. Как можно видеть, на 4-ядерном CPU Intel Core i7-3770 он почти в 4 раза медленнее специализированного алгоритма 4.5 в TPT3. В то же время на 64- и 72-ядерных центральных процессорах его быстродействие сравнимо с 4.5, а алгоритмы 4.8, 4.9 работают в несколько раз быстрее него. При этом, как было сказано, он требует выделения большого количества дополнительной памяти и не сохраняет относительный порядок следования переносимых частиц с одинаковым `ir`.

Таким образом, на CPU этот алгоритм нельзя назвать оптимальным по производительности и выделяемой памяти, так что можно порекомендовать использовать вместо него существенно превосходящие его по этим характеристикам 4.5, 4.8 и 4.9.

**4.13. Ранее использовавшийся в TPT3 алгоритм сортировки частиц на GPU.** Рассматривается тот же самый алгоритм, что и в предыдущем подразделе 4.12, только портированный на GPU при помощи языка программирования графических ускорителей NVIDIA CUDA. Его реализация на GPU находится в файле `cuda_sort_no_shared_memory.cu` [21], где приведен вариант этого алгоритма без использования разделяемой памяти CUDA.

На GPU NVIDIA Tesla V100 он показал среднее время выполнения около 220 мс независимо от флагов оптимизации компилятора, что всего в 9 раз медленнее самого быстрого варианта сортировки на GPU 4.16.

**4.14. Ранее использовавшийся в TPT3 алгоритм сортировки частиц на GPU при использовании разделяемой памяти CUDA.** Рассматривается тот же алгоритм, что и в 4.13, но с использованием разделяемой памяти CUDA. Программный код находится в файле `cuda_sort_shared_memory.cu` [21]. Сильно ограниченный объем разделяемой памяти CUDA не позволил выполнить прямой запуск полного

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

массива частиц размером $N_\text{p} = 2 \cdot 10^7$. Поэтому задача запускалась на массивах меньшего размера, а затем время экстраполировалось на исходный размер.

Именно по этой причине результат измерения среднего времени выполнения алгоритма 4.14 следует интерпретировать с осторожностью. Полученные оценки составили 805 мс при `-O3` и 765 мс при `-O0`. Эти значения включены в табл. 1 только как ориентир и не должны восприниматься как полностью сопоставимые с прямыми измерениями скорости счета остальных алгоритмов.

**4.15. Ранее использовавшийся в TPT3 алгоритм сортировки частиц на GPU без использования разделяемой памяти CUDA, распараллеленный при помощи потоков CUDA.** Здесь алгоритм сортировки 4.13 был распараллелен на GPU при помощи потоков CUDA без использования разделяемой памяти. Его программная реализация приведена в файле `cuda_threads_sort.cu` [21]. Экспериментально было найдено, что на GPU NVIDIA Tesla V100 наилучший результат достигается при 16 потоках CUDA.

Среднее время выполнения составило 120 мс как при `-O3`, так и при `-O0`. Таким образом, даже для старой алгоритмической схемы переход к реализации на графическом ускорителе дает существенное увеличение быстродействия. В то же время использование промежуточных массивов для хранения результатов сортировки в каждом подмассиве частиц, как это делалось на ранних этапах разработки программы TPT3, помимо двукратного увеличения объема потребляемой памяти приводит к росту среднего времени счета на 85% при `-O3` и на 43% при `-O0`.

**4.16. Наиболее быстрый алгоритм сортировки частиц на GPU при помощи библиотеки NVIDIA CUDA CUB.** Наиболее быстрым среди рассмотренных вариантов сортировки переносимых частиц оказался вызов `cub::DeviceRadixSort::SortPairs` из библиотеки CUDA CUB [18, 23]. Этот метод относится к библиотечным реализациям поразрядной сортировки по ключам и хорошо соответствует рассматриваемой задаче сортировки структур по короткому целочисленному полю `ir`. В его реализации формируется отдельный массив ключей, после чего `SortPairs` переставляет и ключи, и соответствующие им структуры частиц.

При попытке использовать эту библиотеку возникла проблема несовместимости использовавшихся версий CUDA, библиотеки CUDA Thrust и библиотеки CUDA CUB. Чтобы устранить несовместимость версий CUDA 11.4 и библиотеки CUDA CUB, методом проб и ошибок было найдено, что самой поздней версией библиотеки CUDA CUB, совместимой с версией CUDA 11.4, является релиз библиотеки CUB 1.14, который и был использован при проведении расчетов. Но далее возникла следующая проблема: библиотека CUDA CUB версии 1.14 была несовместима с использовавшейся библиотекой Thrust версии 1.9.5, потому что была гораздо старее. Единственным способом устранения этой ошибки было определить макрос, приведенный в листинге 8. Это позволило устранить ошибку компиляции, возникавшую по причине отсутствия соответствующих заголовочных файлов и определения этого макроса в более старой версии библиотеки CUDA CUB. Для компиляции программы, использовавшей библиотеку CUDA CUB, необходимо было использовать компилятор `nvcc`, а также соответствующие флаги производительности `-O0` и `O3`.

Листинг 8. Макрос, необходимый для исправления ошибки компиляции из-за несовместимости библиотек
Listing 8. The macro needed to fix a compilation error caused by library incompatibility

```
1 #define THRUST_NS_QUALIFIER thrust
```

В листинге 9 показан пример использования `cub::DeviceRadixSort::SortPairs`. Сортировка здесь проводилась прямо в исходном массиве, без копирования в выходной массив, и из соображений большей ясности для массива частиц использовалось другое имя, отличное от указанного в листинге 1.

В соответствии с принятой в работе методикой измерений в измеряемый интервал времени счета на GPU включалось только *on-device* время выполнения собственно сортировки. Для алгоритма 4.16 к моменту начала измерений на GPU уже были размещены массив ключей и массив значений, а также был определен требуемый объем вре́менного буфера. Поэтому измеряемый интервал характеризует именно время вызова `cub::DeviceRadixSort::SortPairs` для уже подготовленного представления *ключ/значение*. Выделение памяти, подготовка буферов на host, переносы данных между host и device, warm-up и служебная синхронизация вне интервала измерения во время счета не включались. Необходимо сказать, что дополнительная временна́я задержка, связанная с однократным копированием массива ключей

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

Листинг 9. Наиболее быстрый алгоритм сортировки частиц на GPU при помощи библиотеки NVIDIA CUDA CUB

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

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

с CPU на GPU и обратно при многократном вызове процедуры сортировки находящихся на GPU данных пренебрежимо мала и не приводит к снижению производительности при большом числе вызовов сортировки на графическом ускорителе. В рассматриваемой постановке задачи этот вариант показал лучшие результаты: 27 мс при `-O0` и 24 мс при `-O3` на GPU NVIDIA Tesla V100.

С практической точки зрения это означает, что при сценарии, в котором данные уже находятся в памяти графического ускорителя и требуется их многократная сортировка, библиотечная поразрядная сортировка CUB является наиболее предпочтительным вариантом из всех рассмотренных реализаций на GPU.

**5. Обсуждение результатов.** Полученные результаты согласуются с теоретическими ожиданиями для сортировки структур по короткому целочисленному ключу. Поскольку сортировка выполняется по целочисленному ключу мощности $K = 5$, универсальные алгоритмы сортировки, основанные на сравне-

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

ниях, оказываются в менее выгодном положении, чем специализированные схемы, такие как сортировка подсчетом и поразрядная сортировка. Это особенно заметно при сравнении `std::sort` из 4.1 с алгоритмами 4.5, 4.6 и 4.16.

Среди библиотечных вариантов сортировки на CPU наиболее сильным решением оказался `boost::sort::spreadsort::integer_sort` из 4.2. Он заметно быстрее `std::sort`, что согласуется с его гибридной природой и лучшим соответствием задаче сортировки по целочисленному ключу. Однако в рассматриваемой постановке задачи даже этот вариант уступает лучшим специализированным реализациям алгоритма сортировки на CPU.

На центральных процессорах лучший результат зависит от архитектуры и от требований к устойчивости. На 4-ядерном CPU Intel Core i7-3770 наиболее удачным оказался специализированный авторский устойчивый алгоритм TPT3 4.5 со средним временем выполнения 140 мс. На 64-ядерном CPU Intel Xeon Gold 6242 лучшие средние значения времени показали схемы с параллельной сортировкой подмассивов 4.8 и 4.9 — около 90 мс. Это означает, что при переходе к многоядерным платформам возрастает роль инженерных аспектов программной реализации: распараллеливания, поведения памяти и накладных расходов на перемещение структур переносимых частиц.

Алгоритм 4.5 остается важным результатом именно потому, что сочетает высокую производительность с устойчивостью. Его быстродействие на многоядерных CPU ограничивается в основном пропускной способностью памяти, однако на CPU всего с 4 ядрами он показал один из лучших результатов среди устойчивых вариантов сортировки на центральных процессорах. Следовательно, специализированная схема, основанная на алгоритме сортировки подсчетом, имеет ценность не только как теоретически уместный метод для малого диапазона ключа, но и как практически эффективная реализация.

На GPU лидером с заметным отрывом оказался алгоритм `cub::DeviceRadixSort::SortPairs` 4.16: 24 мс при `-O3` на NVIDIA Tesla V100. Ближайшим к нему вариантом сортировки стала реализация на GPU специализированного авторского алгоритма сортировки частиц в TPT3 при помощи CUDA Thrust 4.6 со средним временем счета 41 мс. Прямой вызов `thrust::sort` 4.7, хотя и уступает лидеру производительности, все же обеспечивает высокую скорость расчетов при минимальных трудозатратах на программирование.

Следует отдельно подчеркнуть, что сравнение вариантов программного кода на GPU в данной статье проводится только по *on-device* времени сортировки. Для алгоритма 4.16 это означает сравнение именно этапа `SortPairs` при уже подготовленном на устройстве представлении *ключ/значение*. В рамках принятого критерия оптимальности такое сопоставление является корректным и согласованным с общей методикой измерений. Настоящая работа не ставила целью сравнение полного *end-to-end* времени счета с учетом выделения памяти и переносов данных между host и device. Поэтому рекомендация использовать алгоритм 4.16 относится прежде всего к задачам, подобным сортировке частиц в программе TPT3, где данные длительное время хранятся в памяти GPU и алгоритм сортировки вызывается многократно.

Результат скорости счета алгоритма 4.14 не следует интерпретировать наравне с прямыми измерениями производительности остальных алгоритмов, поскольку он получен экстраполяцией. Напротив, результаты сравнения быстродействия алгоритмов 4.13 и 4.15 полезны с точки зрения программной инженерии: они показывают, насколько сильно качество программной реализации на GPU может повлиять на итоговую вычислительную производительность даже при одной и той же базовой идее алгоритмов.

Большой интерес представляло исследование зависимости скорости счета алгоритмов, использующих для сортировки подмассивы, от их числа $N_{\text{bin}}$. С этой целью из алгоритмов, использующих для распараллеливания процесса сортировки подмассивы, были выбраны самые быстрые: 4.5, 4.8 и 4.9 на центральных процессорах и 4.6 и 4.10 (как полный аналог 4.9 на GPU) на графических ускорителях. Полученные в результате значения времени сортировки массива из $N_{\text{p}} = 2 \cdot 10^7$ переносимых частиц в зависимости от числа подмассивов $N_{\text{bin}}$ при помощи алгоритмов 4.5 на CPU и 4.6 на GPU представлены на рис. 1, а при помощи 4.8 и 4.9 на CPU, а также 4.10 на GPU — на рис. 2.

Из рис. 1 видно, что вначале при $N_{\text{bin}}$ не более 100–1000 (в зависимости от рассматриваемого CPU) наблюдается увеличение времени счета. Видно, что оно имеет место, когда число частиц в подмассивах велико, т.е. превышает десятки тысяч для многоядерных CPU и сотни тысяч для Intel Core i7-3770. Начиная с $N_{\text{bin}}$=100 для 4-ядерного центрального процессора и с $N_{\text{bin}}$=1000 для многоядерных CPU, время счета достигает минимума и при дальнейшем увеличении числа подмассивов остается неизменным. Для многоядерных центральных процессоров это установившееся значение времени меньше времени счета при $N_{\text{bin}}$=1 в 2.5 раза, но для 4-ядерного Intel Core i7-3770 оба значения почти совпадают. При этом произво-

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

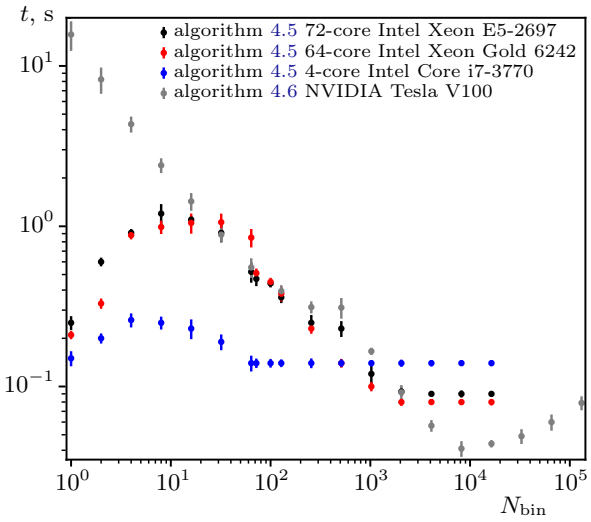


Рис. 1. Время счета сортировки массива размером $2 \cdot 10^7$ частиц при помощи алгоритмов 4.5, 4.6 в зависимости от величины числа подмассивов $N_{\text{bin}}$ на различных вычислительных платформах

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 $N_{\text{bin}}$ on different computing platforms

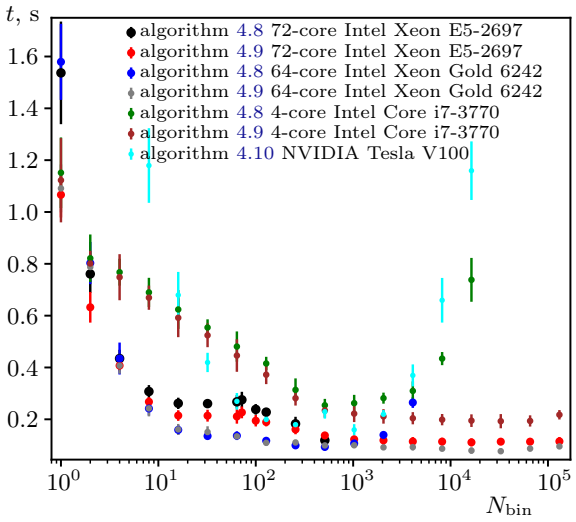


Рис. 2. Время счета сортировки массива размером $2 \cdot 10^7$ частиц при помощи алгоритмов 4.8–4.10 в зависимости от величины числа подмассивов $N_{\text{bin}}$ на различных вычислительных платформах

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 $N_{\text{bin}}$ on different computing platforms

дительность расчетов на GPU NVIDIA Tesla V100 растет прямо пропорционально числу подмассивов от $N_{\text{bin}} = 1$ до примерно $N_{\text{bin}} = 20$ и, начиная с этой величины, приблизительно пропорционально $\sqrt{N_{\text{bin}}}$, достигая максимума при $N_{\text{bin}}$=8192 (использование этой величины для алгоритма сортировки 4.6 отражено в табл. 1), после чего наблюдается незначительное замедление скорости счета.

Зависимость времени счета от $N_{\text{bin}}$ при использовании алгоритмов сортировки 4.8 и 4.9 на различных центральных процессорах и 4.10 на GPU NVIDIA Tesla V100 показана на рис. 2. Измерения скорости счета алгоритма 4.8 заканчиваются при $N_{\text{bin}}$, существенно меньших, чем для 4.9, поскольку на использованных вычислительных платформах присутствовало системное ограничение на число создаваемых вычислительных потоков, которое не позволяло продвинуться в область бóльших $N_{\text{bin}}$. Тем не менее, на примере результатов для 4-ядерного CPU на рис. 2 можно предположить, что исследуемое время счета алгоритма сортировки 4.8 с ростом числа вычислительных потоков (и, соответственно, подмассивов) уменьшается, достигая минимума при $N_{\text{bin}} \sim 1000$, а затем опять переходит к росту. В то же время использование OpenMP для распараллеливания алгоритма сортировки 4.9 приводит к непрерывному увеличению скорости счета с ростом $N_{\text{bin}}$, достигающей максимума при $N_{\text{bin}} \sim 2 \cdot 10^3$ и далее остающейся неизменной. Поэтому, исходя из рис. 2, можно с хорошей точностью считать, что максимальная производительность алгоритмов 4.8, 4.9 на центральных процессорах наблюдается при $N_{\text{bin}} = 512$ и 2048 соответственно, что и отражено в табл. 1. Для алгоритма 4.10 на GPU NVIDIA Tesla V100 максимум производительности при сортировке $N_{\text{p}} = 2 \cdot 10^7$ частиц достигается при $N_{\text{bin}} \sim 200$, и зависимость времени счета от числа подмассивов имеет еще более выраженный характер, достигая 7.7 с при $N_{\text{bin}} = 1$ и 2 с при $N_{\text{bin}} = 32768$.

На рис. 3 показано приведенное время сортировки на 1 частицу в зависимости от величины $N_{\text{p}}$ алгоритмов 4.5 на CPU, а также 4.6 и самого быстрого алгоритма сортировки 4.16 на GPU, которое аппроксимировалось единым образом: $t/N_{\text{p}} = t_1/N_{\text{p}} + t_2$, где $t$ — полное время счета, $t_1$ — время инициализации программного кода на рассматриваемой вычислительной платформе, $t_2$ — приведенное время счета на 1 частицу после выхода вычислений на оптимальный интенсивный режим работы с высокой эффективностью распараллеливания расчетов, когда временны́е затраты на инициализацию становятся пренебрежимо малыми. Приведенное время счета алгоритма 4.5 на 64-ядерном CPU Intel Xeon Gold 6242 (черные точки) описывалось фитирующей функцией $5.71 \cdot 10^7/N_{\text{p}} + 1.51$ нс, алгоритма 4.6 на GPU — $2.05 \cdot 10^6/N_{\text{p}} + 1.98$ нс (красные точки), а 4.16 на GPU — $5.46 \cdot 10^5/N_{\text{p}} + 1.21$ нс (синие точки). На GPU NVIDIA Tesla V100 максимальный размер массива частиц, для которого были проведены вычисления,

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

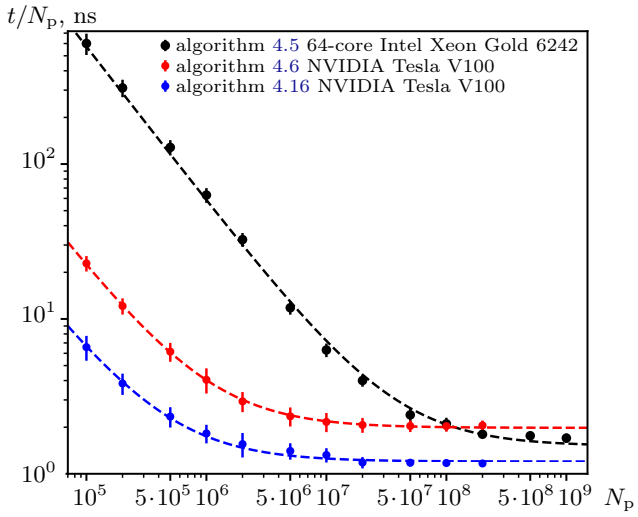


Рис. 3. Приведенное время счета на 1 частицу алгоритмов сортировки 4.5 на CPU и 4.6 и 4.16 на GPU при $N_\mathrm{bin} = 2048$

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

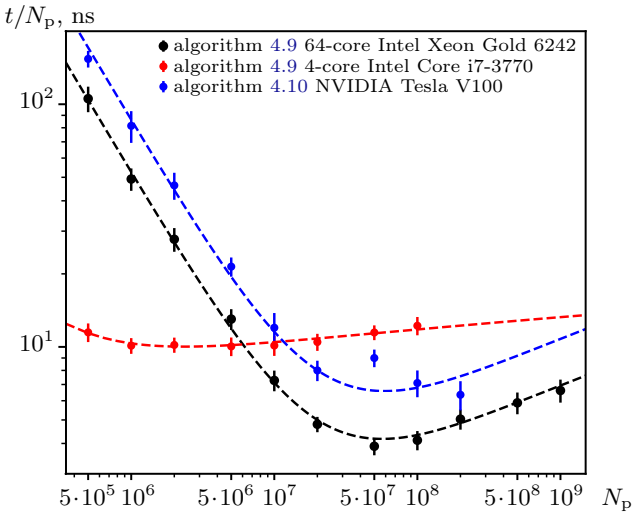


Рис. 4. Приведенное время счета на 1 частицу алгоритмов сортировки 4.9, 4.10 на различных вычислительных платформах при $N_\mathrm{bin} = 2048$

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

составляет $N_\mathrm{p} = 2 \cdot 10^8$, поскольку при таком $N_\mathrm{p}$ суммарный объем памяти, выделяемый под входной и выходной массивы частиц в компьютерной памяти, составляет примерно 21 ГБ, а при $N_\mathrm{p} = 5 \cdot 10^8$ он уже превышает объем памяти графического ускорителя, равный 32 ГБ. Таким образом, самое большое время инициализации наблюдается для алгоритма сортировки 4.5 на 64-ядерном CPU Intel Xeon Gold 6242 и составляет 57 мс, а для алгоритмов 4.6 и 4.16 на GPU — 2 мкс и 0.5 мкс соответственно. По приведенному времени счета на 1 частицу алгоритм 4.5 на многоядерном CPU на 25% быстрее 4.6 на GPU и, в свою очередь, примерно на столько же медленнее 4.16.

Наконец, на рис. 4 приводится зависимость от величины $N_\mathrm{p}$ времени счета алгоритмов 4.9 на CPU и 4.10 на GPU. В этих алгоритмах параллельно в каждом подмассиве используется стандартная сортировка `std::sort`, имеющая асимптотическую временну́ю сложность в среднем $O(n \log n)$, где $n$ — число сортируемых элементов, так что по аналогии с рис. 3 полное время счета может быть представлено в виде: $t = t_1 + t_2 \cdot N_\mathrm{p} \log N_\mathrm{p}$. В связи с этим приведенное время сортировки алгоритмом 4.9 на 1 частицу на 4-ядерном центральном процессоре Intel Core i7-3770 описывалось функцией $1.5 \cdot 10^6/N_\mathrm{p} + 0.64 \cdot \log N_\mathrm{p}$ нс (красная кривая). При этом максимальное значение числа частиц на этом CPU, при котором были проведены вычисления, составило $N_\mathrm{p} = 10^8$. При таком значении $N_\mathrm{p}$ объем памяти, выделяемой для хранения входного массива частиц, составляет 5.2 ГБ, а для входного и выходного массивов — уже 10.4 ГБ, при этом объем оперативной памяти на 4-ядерном центральном процессоре Intel Core i7-3770 составляет всего 8 ГБ. Так что работоспособность вычислительного устройства в данном случае, очевидно, поддерживается за счет механизма подкачки и swap-памяти, размер которой на этом CPU — 11 ГБ. Но при увеличении $N_\mathrm{p}$ в 2 раза не хватает уже и swap-памяти, вычислительное устройство зависает, с чем и связано верхнее ограничение на $N_\mathrm{p}$ на 4-ядерном CPU. Однако на 64-ядерном CPU Intel Xeon Gold 6242 и на GPU NVIDIA Tesla V100 не удавалось описать время счета данной модельной зависимостью, и оно было профитировано соответственно как $5.17 \cdot 10^7/N_\mathrm{p} + 1.8 \cdot 10^{-6} \cdot \log^5 N_\mathrm{p}$ нс (алгоритм 4.9, черная кривая) и $8.5 \cdot 10^7/N_\mathrm{p} + 2.8 \cdot 10^{-6} \cdot \log^5 N_\mathrm{p}$ нс (алгоритм 4.10, синяя кривая), в связи с чем не представляется возможным интерпретировать физический смысл параметров в этих зависимостях. В то же время из рис. 4 можно выяснить, что минимальное приведенное время сортировки алгоритмом 4.9 на 1 частицу на 4-ядерном CPU составило 10 нс, на 64-ядерном CPU — 4 нс, а при использовании алгоритма сортировки 4.10 на GPU NVIDIA Tesla V100 — 6.7 нс, что в несколько раз больше, чем для алгоритмов на рис. 3.

В целом, полученные данные подтверждают, что в рассматриваемой постановке не существует единственного “лучшего” алгоритма без оговорок. Для CPU выбор зависит от архитектуры и требования

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321

устойчивости, а для GPU — от того, сравнивается ли *on-device* или *end-to-end* время. Хотя, как уже было сказано, при длительном нахождении данных в памяти GPU и многократном вызове процедуры сортировки дополнительная временна́я задержка, связанная с однократным копированием данных с CPU на GPU и обратно, пренебрежимо мала и определяющее значение имеет именно время счета *on-device*. Тем не менее, в рамках принятого в работе критерия оптимальности можно сделать вполне определенные выводы для заданных структуры данных и диапазона ключей.

**6. Заключение.** В работе сравнивались 16 реализаций сортировки массива из $N_\mathrm{p} = 2 \cdot 10^7$ переносимых частиц, описываемых структурой в листинге 1, по целочисленному полю `ir`. Для всех алгоритмов использовался один и тот же входной массив со случайными значениями `ir` из множества $\{-1, 0, 1, 2, 3\}$.

Показано, что при столь малом диапазоне ключа универсальные алгоритмы сортировки, основанные на сравнениях, не являются наилучшим выбором по производительности. Среди реализаций на центральных процессорах наилучшие результаты показали специализированный авторский устойчивый алгоритм сортировки в TPT3 4.5 на 4-ядерном CPU Intel Core i7-3770 (140 мс) и многоядерных центральных процессорах (80 и 90 мс на 64- и 72-ядерных CPU) и схемы с параллельной сортировкой подмассивов 4.8–4.9 на 64-ядерном CPU Intel Xeon Gold 6242 ($\sim$ 90 мс). Среди библиотечных вариантов сортировки на центральных процессорах наилучшим по производительности оказался `boost::sort::spreadsort::integer_sort`, однако он уступает алгоритмам-лидерам быстродействия.

На GPU NVIDIA Tesla V100 самым быстрым оказался алгоритм 4.16 на основе метода поразрядной сортировки из библиотеки CUDA CUB: 24 мс при `-O3`. Это значение относится именно к этапу сортировки при уже заранее подготовленных данных на GPU. Ближайшим к нему по производительности стал алгоритм 4.6, представляющий собой реализацию на GPU специализированной схемы сортировки в TPT3 4.5 при помощи библиотеки CUDA Thrust, со временем счета 41 мс.

Таким образом, в рамках принятого в работе критерия оптимальности наиболее предпочтительным вариантом для рассматриваемой постановки задачи на GPU является поразрядная сортировка из библиотеки CUB, а для CPU — специализированные или специально распараллеленные программные реализации, адаптированные к структуре данных и малому диапазону ключа. Эти выводы относятся именно к рассмотренной структуре переносимой частицы и сравнению производительности по *on-device* времени вычислений на GPU. Перенос их на другие типы данных и диапазоны ключей требует отдельной экспериментальной проверки.

## Список литературы


1. *Bergmann R.M., Vujić J.L.* Algorithmic choices in WARP — A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs // Annals of Nuclear Energy. 2015. **77**. 176–193. doi 10.1016/j.anucene.2014.10.039.
2. *Liu C.* Study on the particle sorting performance for reactor Monte Carlo neutron transport on Apple Unified Memory GPUs // EPJ Web of Conferences. 2024. **302**. Article Number 04001. doi 10.1051/epjconf/202430204001.
3. *Tramm J., Romano P., Shriwise P., et al.* Performance portable Monte Carlo particle transport on Intel, NVIDIA, and AMD GPUs // EPJ Web of Conferences. 2024. **302**. Article Number 04010. doi 10.1051/epjconf/202430204010.
4. *Egorova M.S., Dyachkov S.A., Parshikov A.N., Zhakhovsky V.V.* Parallel SPH modeling using dynamic domain decomposition and load balancing displacement of Voronoi subdomains // Comp. Phys. Comm. 2019. **234**. 112–125. doi 10.1016/j.cpc.2018.07.019.
5. *Вшивков В.А., Маркелова Т.В., Шелехов В.И.* Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ. 2008. **33**, № 4. 79–92. https://cyberleninka.ru/article/n/ob-algoritmah-sortirovki-v-metode-chastits-v-yacheykah. Дата обращения: 7 июня 2026.
6. *Романенко А.А., Снытников А.В.* Оптимизация переупорядочивания модельных частиц при реализации метода частиц в ячейках на GPU // Вестник НГУ. Серия: Информационные технологии. 2019. **17**, № 1. 82–89. doi 10.25205/1818-7900-2019-17-1-82-89.
7. *Snytnikov N.V., Stoyanovskaya O.P., Glushko T.A.* Parallel algorithm for supercomputing simulation of dust-gaseous gravitating systems using particle-in-cell and SPH methods // Journal of Physics: Conference Series. 2019. **1336**, N 1, Article Number 012021. doi 10.1088/1742-6596/1336/1/012021.
8. *Agostinelli S., Allison J., Amako K., et al.* Geant4 — a simulation toolkit // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2003. **506**, N 3. 250–303. doi 10.1016/S0168-9002(03)01368-8.

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321


9. *Allison J., Amako K., Apostolakis J., et al.* Geant4 developments and applications // IEEE Transactions on Nuclear Science. 2006. **53**, N 1. 270–278. doi 10.1109/TNS.2006.869826.
10. *Галюзов А.А., Косов М.В.* Увеличение производительности моделирования радиационного переноса при помощи параллельной программы Toolkit for Particle Transport 3 // Программные продукты и системы. 2025. **38**, № 2. 269–279. doi 10.15827/0236-235X.150.269-279.
11. *Галюзов А.А., Косов М.В.* Программа для высокопроизводительного моделирования переноса радиации ТРТ3. Свидетельство о государственной регистрации программы для ЭВМ 2024614877. Дата регистрации 29.02.2024.
12. Стандарт препроцессорных директив OpenMP. https://www.openmp.org. Cited June 9, 2026.
13. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda/toolkit. Cited June 7, 2026.
14. Стандарт препроцессорных директив OpenACC. https://www.openacc.org. Дата обращения: 7 июня 2026.
15. *Knuth D.E.* The art of computer programming. Vol. 3. Sorting and Searching. 2nd ed. Boston: Addison-Wesley, 1998.
16. *Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.* Introduction to algorithms. 3rd ed. Cambridge: MIT Press, 2009.
17. *Ross S.J.* The spreadsort high-performance general-case sorting algorithm // Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. 2002. **3**. 1100–1106.
18. *Satish N., Harris M., Garland M.* Designing efficient sorting algorithms for manycore GPUs // 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. *Musser D.R.* Introspective sorting and selection algorithms // Software: Practice and Experience. 1997. **27**, N 8. 983–993.
20. *Галюзов А.А., Косов М.В.* Расчет выхода гамма-квантов при делении $^{(235)}$U в урановом кубе программой ТРТ3 // Технологии ЭМС. 2024. **89**, № 2. 26–32.
21. Репозиторий с исходным кодом примеров из статьи. 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. Репозиторий с исходным кодом библиотеки CUDA CUB. https://github.com/NVIDIA/cub.git. Дата обращения: 7 июня 2026.



| *Получена* | *Принята* | *Опубликована* |
|---|---|---|
| *30 апреля 2026 г.* | *24 мая 2026 г.* | *1 июля 2026 г.* |


### Информация об авторе

*Андрей Андреевич Галюзов* — ст. науч. сотр.; Всероссийский научно-исследовательский институт автоматики имени Н. Л. Духова (ВНИИА), ул. Сущевская, 22, 127030, Москва, Российская Федерация.

### 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.
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.

2026, **27** (3), 317–336. doi 10.26089/NumMet.v27r321


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.