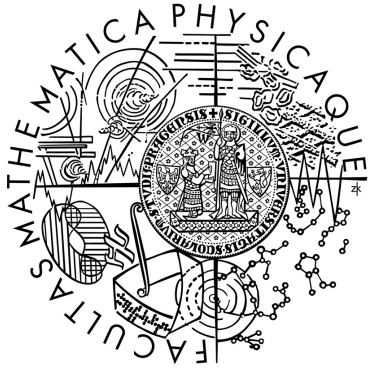
Charles University in Prague
Faculty of Mathematics and Physics

# MASTER THESIS




Bc. Radim Vansa

## Parallel Data-processing on GPGPU

Department of Software Engineering

Supervisor of the master thesis: RNDr. Martin Kruliš

Study programme: Informatics
Specialization: I2 Software Systems
Prague 2012

I would like to give thanks to my supervisor RNDr. Martin Kruliš for his time and support. He was always glad to offer a help with both theoretical part of this thesis and faulty hardware or libraries required for completion of this thesis.
My thanks to my parents and family, whose support and guidance have helped me sail through difficult chapters of my life.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ….....

# Anotace

Název práce: Paralelní zpracování dat na GPGPU

Autor: Bc. Radim Vansa

Katedra / Ústav: Katedra Softwarového Inženýrství

Vedoucí diplomové práce: RNDr. Martin Kruliš, Katedra Softwarového Inženýrství

Abstrakt: Dnešní grafické karty mohou sloužit nejen pro vykreslování 3D obrazu, ale prostřednictvím frameworků jako např. OpenCL umožňují využít sílu mnoha výpočetních jader k obecnějšímu zpracování velkého množství informací. Tato práce se soustředí na základní operace používané v databázových systémech, konkrétně na třídění a hledání průniku množin. Nabízí několik postupů řešení každého z těchto problémů a hodnotí výsledky implementací těchto algoritmů. Ukazuje se, že obě zmíněné úlohy mohou být úspěšně řešeny s využitím grafických karet, a to se značným urychlením oproti tradičnímu přístupu s výpočty pouze na vícejádrovém CPU.

Klíčová slova: paralelní, GPU, OpenCL, třídění, průnik množin

# Annotation

Title: Parallel data-processing on GPGPU

Author: Bc. Radim Vansa

Department / Institute: Department of Software Engineering

Supervisor of the master thesis: Martin Kruliš, M.Sc., Department of Software Engineering

Abstract: Modern graphic cards are no longer limited to 3D image rendering. Frameworks such as OpenCL enable developers to harness the power of many-core architectures for general-purpose data-processing. This thesis is focused on elementary primitives often used in database management systems, particularly on sorting and set intersection. We present several approaches to these problems and evalute results of benchmarked implementations. Our conclusion is that both tasks can be successfully solved using graphic cards with significant speedup compared to the traditional applications computing solely on multicore CPU.

Keywords: parallel, GPU, OpenCL, sorting, set intersection

# Table of Contents

# 1. Introduction

Several years ago a limit in frequency of processors was reached and also the instruction level parallelism could not be effectively widened anymore. That is why the focus has moved to multi-core CPUs and software developers had to adapt to this hardware. Applications started to use multiple threads with all the advantages (speedup) and disadvantages such as the need of complicated synchronization. In the latest years the high-performance computing research was focused on inter-thread communication, synchronization such as lock-free algorithms, transactional memory or load balancing, and efficient use of CPU caches.

Nevertheless, the amount of CPU cores still grows and the well-known patterns such as transparent caches and uniformly accessible memory are no longer scalable. A new pattern of many-core architecture with different programming model emerged from the area of specialized single-purpose hardware used for an acceleration of graphical computations.

Modern graphics cards are no longer limited to the execution of hardwired operations designed for 3D rendering, but also allow a parallel processing of non-graphical data. This new architecture can yield higher performance than conventional CPUs, for certain applications even in orders of magnitude [1]. In contrast with its name 'general-purpose' GPUs (GPGPUs) this kind of hardware has lower performance in some general tasks in comparison with CPU cores, due to differences in architecture. One may consider them rather as co-processors suitable for heavy-computation than a substitute for CPUs.

Since database management systems often need to process a huge amounts of data, researchers find here an opportunity to offload some parts of the computation to the GPGPUs ([2], [3]). This is also the main objective of this thesis. We study primitives used in database operations, particularly the sorting and set intersection, examine the performance of our implementation and compare it to the performance of sequential or parallel algorithms running on a few CPU cores.

We study two different problems – sorting and set intersection. That is why we use rather unusual structure of the thesis. At first we describe attributes common to both problems, the GPGPU architecture and OpenCL programming model in section 2, and our benchmarking methodology in section 3. In following two chapters we

study each problem – sorting in section 4 and set intersection in section 5. In these sections, we describe implemented algorithms solving the problem, provide results of their benchmarks and comparison with standard CPU approach, and make suggestions for future research. Conclusive remarks and comments are contained in section 6.

# 2. GPGPU Programming

This chapter describes the architecture of GPGPUs and how it is mapped into the OpenCL framework. As there are currently two major manufacturers of GPUs, ATI Technologies, Inc. (owned by Advanced Micro Devices, Inc.) and NVIDIA Corporation, the terminology and architecture overview will be provided for their products.

## 2.1 GPGPU Architecture Overview

High-performance GPGPU can hold up to several hundreds of stream processors. These processors are much simpler than ordinary cores in CPUs – they do no instruction reordering, and the instruction execution speed is predictable and fixed to the GPU frequency. GPU runs no operating system, memory is addressed directly without any paging or segmentation mechanism, and there is no need for interrupt handling. This allows narrower instruction set and therefore less complicated hardware.

These stream processors are not completely independent as on CPU, where each core can execute a thread with a different code on a different part of memory. The stream processors are grouped into multiprocessors with single instruction decoding unit. These groups are called *warps* on NVIDIA GPUs with 32 stream processors and *wavefronts* on ATI GPUs with 16, 32 or 64 processors (depending on the model).

Each thread in the group has private registers and stack but the program counter is shared. If the program flow control diverges within this group, all branches are executed serially, causing a great performance hit and sometimes even deadlock in the program. Understanding this concept, which is called 'single instruction multiple threads' (SIMT), is one of the most important things in GPU programming.

Each stream processor executes only single thread at one moment but more threads can be scheduled for execution. The multiprocessor has limited number of registers and these are used also by threads that are scheduled but not currently running – the thread state (register contents) is not transferred off the multiprocessor. This allows fast context-switch performed entirely in the hardware. If the execution

must wait for slow memory access then the hardware simply switches to another group of threads that can execute hiding the memory latency.

Similarly as on CPUs there is a hierarchical memory structure:

- The global GPU memory has several hundreds of megabytes or few gigabytes, and it is separated from the common main memory[1] (RAM). Data transfers between GPU global memory and RAM are issued from the host (CPU) program code.
- The GPU may feature a transparent L2 cache shared between all multiprocessors.
- Several other memory types can be directly on the multiprocessor chip:
  - Transparent L1 cache speeding up access to the global memory.
  - Shared memory for communication between the stream processors. Transfers between shared memory and global memory are controlled directly from the code executed on GPU.
  - Constant cache for repeatedly read non-modifiable data.
  - Texture cache for image data optimized for 2D spatial locality.

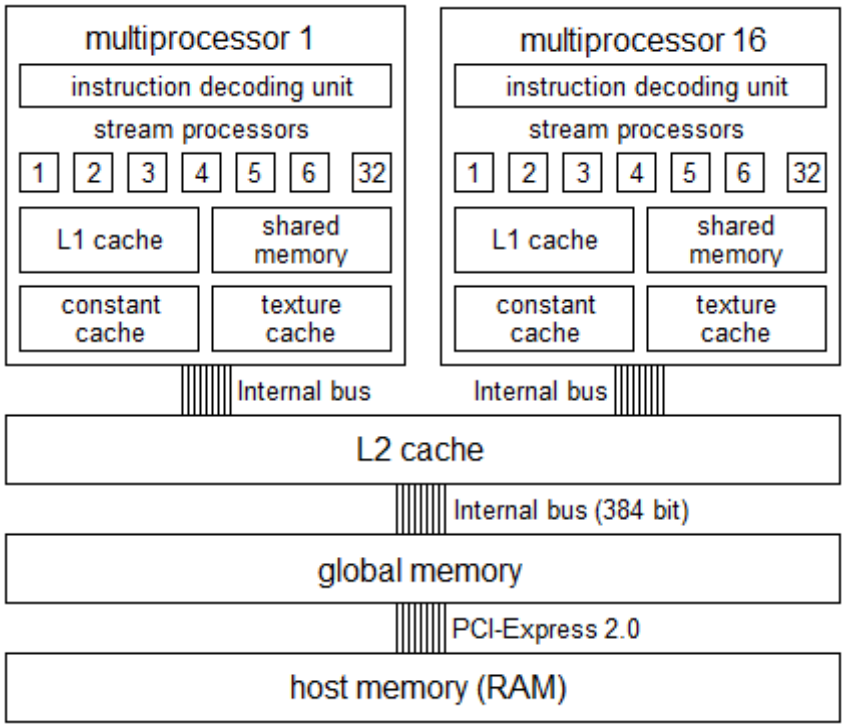



Image 2.1: NVidia GeForce GTX 580 architecture

1 In technical documentation GPU is often referred to as the *device* and CPU as the *host* – therefore, the common memory is referred to as the *host memory*.

## 2.2 OpenCL

OpenCL [4] is an open standard for "cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices" [5]. Therefore, it is not limited to GPGPUs but the same code should work[2] also on multi-core CPUs or on Cell Broadband Engine.

The programming model in OpenCL framework considers two parts of the program: the host code written in any language with bindings to the particular CL library (C++ in case of this thesis) and the device code which will run on the massively parallel device. This device code should be written in the OpenCL language, which is a subset of ISO C99 with some extensions. Its limitations are for example prohibition of function pointers or recursive function calls. Examples of extensions to the ISO C99 are built-in vector data-types and functions or address space and data alignment attributes.

### 2.2.1 Execution Model

The CPU (called *host*) communicates with the *device* (GPU in our case) through a *command queue*. The OpenCL program executed on the device consists of functions (or rather procedures) called *kernels*. These kernels are not called directly from the host program but commands to execute them are sent to the command queue. As the device program cannot access the host memory (RAM) the host program must provide it through enqueuing a command to copy the data from host memory to device memory to the command queue and then collect the results using another command to copy the data in the opposite direction. For these operations the memory is encapsulated in *buffer objects*.

The queues are in-order by default – the commands are executed in the same order as they are enqueued, and at each moment only one command can be executed. Out-of-order command queues can execute multiple commands in parallel if there is enough hardware resources. *Events* are used for synchronization. Each command can depend on multiple events, when a command is finished the event associated with it is raised and the device starts executing another command(s) with satisfied

2 This means the code should be functional, but the performance may vary a lot. See Section 2.3 for details.

dependencies. One device can run multiple command queues – events can be used also for a synchronization of commands in different queues.

Threads are called *work-items*. In the host program, the number of threads required for the task is specified as a range in one-, two- or three-dimensional space. Each work-item queries its identification number[3] and according to it selects the data that it should process. For example, in a matrix-multiply algorithm the identifier could be the position in output matrix, and each work-item should compute the scalar product of the two corresponding vectors from input matrices and store it on this position.

Work-items are grouped into *work-groups*. Work-items within a work-group can synchronize using *barrier* commands – no work-item can cross the barrier statement until all work-items in the same work-group reach it. The local memory is shared by work-items in a single work-group and is not accessible from any other work-group. Work-items from different work-groups can communicate only through atomic operations in the global memory. However, there is no guarantee that different work-groups will be executed in parallel[4]. Therefore there cannot be anything like work-group-wide barrier.

### 2.2.2 Memory Model

The largest and the slowest memory is the *global* memory. This is shared by all work-items and available for both read and write access. It is also accessible by the host through the command queue and persists across different kernel executions. However, the memory model is relaxed, and therefore the programmer must explicitly cast a memory fence when a consistent view is required.

*Constant* memory is a small part of memory with read-only access. The reads from this memory are cached in separate constant memory cache.

Another memory type, usually located on the multiprocessor chip, is called *local*[5] memory. It is accessible only by work-items within a single work-group and it

3 This identifier is a position in multidimensional space as specified on the host.

4 The work-group runs non-preemptively on single multiprocessor. Other work-groups cannot execute on this multiprocessor until this work-group finishes.

5 In NVidia documentation OpenCL *local* memory is described as *shared* memory. This is rather ambiguous because it also uses the term local memory for a memory with different characteristics which is used for register spilling or private arrays. However, this memory is not located on chip, and therefore, is slower than both the private and shared memory.

is significantly faster than the global memory[6] – it can be thought of as a cache with an explicit access. It is intended for shared variables and communication between work-items. The memory model is relaxed similarly to the global memory, requiring memory fences for synchronization.

Work-item local variables are stored in *private* memory – no other work-item can access it. This memory usually resides in registers. That makes this type of memory the fastest from all available memory types.

The table below shows the sizes and latencies [6] of various memory types on NVidia GTX580 used for experiments in this thesis:

| | |
|---|---|
| Global memory | ~1.5 GB, latency 400-800 cycles, 16 kB L1 cache on each multiprocessor, 768 kB shared L2 cache |
| Constant memory | 64 kB |
| Local memory | 48 kB, latency ~100× smaller than global memory |
| Private memory | 32768 32-bit registers = 128 kB, latency 22-24 cycles for read-after-write |

Both local memory and register count are exclusive for each of the 16 multiprocessors, the global memory is shared.

# 2.3 Performance Considerations

This section describes some design decisions that are not part of the OpenCL specification but highly affect the performance. As we have used NVidia card for the experiments, the behavior will be described for these GPUs.

## 2.3.1 Coalesced Access to Global Memory

Although the work-item can access any part of global memory, the performance may vary according to the access pattern of the whole warp[7] (threads = work-items running on single multiprocessor). Multiple parallel requests to load or write data are coalesced into several 32-, 64- or 128-byte transactions. Moreover, these transactions must be aligned to their size[8].

6 On NVidia GPUs the same hardware is used for local=shared memory and for L1 cache of global memory. In CUDA it is possible to configure which of the two modules (16 kB/48 kB) will be used as the shared memory and which will host L1 cache.

7 In fact on some devices this is only half-warp. See NVidia OpenCL Programming Guide [7] and NVidia OpenCL Best Practices Guide [8] for details. The behaviour on ATI GPUs may also vary.

8 For example 64-byte transaction should start at address divisible by 64, otherwise the request would be split into multiple transactions.

Therefore, the optimal pattern is to access the (*kw* + *i)*-th element from the *i*-th work-item within a warp of size *w* (let us call this the simple access pattern). Older GPUs were not able to coalesce memory accesses with any different pattern; newer ones can handle any access pattern with the minimum number of transactions. Nevertheless, using only single memory transaction is always optimal.

This is why the data layout should be organized as a structure-of-arrays rather than array-of-structures more common in CPU programming[9]. For example, if we had an array of key-value pairs, accessing keys would introduce two memory transactions transferring also the values, therefore, wasting the memory bandwidth. This access pattern is called *strided*. You can imagine that with completely random memory access patter, there would be a separate memory transaction for each access to the memory, causing heavy impact on the performance of the program.

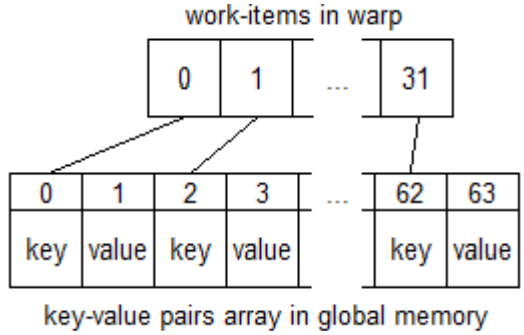



Figure 2.2: Strided memory access pattern

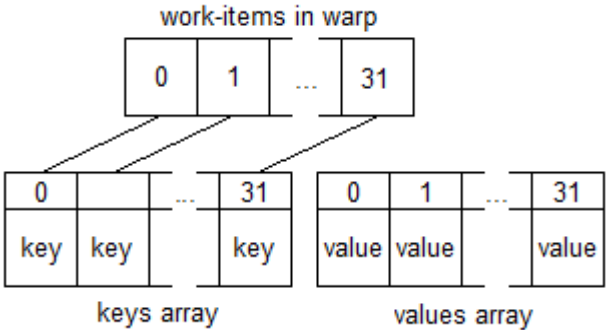



Figure 2.3: Coalesced memory access

Similar problem as the strided access is misaligned access. This would also require more than one memory transaction to satisfy the request.

All these misappropriates can be softened by global memory caches (L1 on multiprocessor and L2 shared) but should be avoided if possible.

9 Array-of-structures data organization is sometimes popular even in CPU programming, particularly in column-oriented database management systems.

### 2.3.2 Bank Conflicts in Local Memory

Local (or "shared" in NVidia terminology) memory is organized into memory modules called banks. If a half-warp accesses *n* memory addresses within *n* different memory banks the requests are processed simultaneously. However, if some *bank-conflict* (concurrent access to one bank from two work-items) occurs, all the requests to access this bank must be serialized, decreasing the throughput. The only exception is a *broadcast* when all requests to the memory bank read from single address – in this case the data can be broadcasted and the request is processed in a single step.

In the local address space, successive 32-bit words belong to successive banks, therefore with 16 memory banks the *i*-th and *(i+16)*-th addresses belong to the same bank. The strided access from a half-warp as described in previous section would therefore cause two requests on each of the 8 banks and 8 banks would stay idle. On the other hand, unlike in global memory the misaligned access is not a problem.

### 2.3.3 Differences Across GPU Vendors

Although the architectures of ATI and NVIDIA GPUs are similar in the essence, there are many small differences. The GPU program source code in OpenCL (described in next part) can run pretty well on GPU from either of the manufacturers, for obtaining the maximum performance the code must be "tuned" for the particular one. The common performance boost is said to be about 10 – 20%, but the result may vary much more [9]. One reason for all are ATI's VLIW5 or VLIW4 vector processors. These run better with vector data-types as the processor has multiple ALU units. Using vector data-types on NVidia is unnecessary and may be even counterproductive.

# 3. Benchmarking Methodology

Since we study two different problems via single methodology, our approach is summarized in this chapter in advance. In the section below we describe what and how we measure and, at last, problems we encountered. Described methodology is used in the following chapters where both problems are treated separately.

## 3.1 Comparing GPUs

Comparing GPUs versus CPUs is generally problematic. The speedup cannot be evaluated directly as with an improved algorithm, because it depends on the additional hardware with varying attributes.

Manufacturers assess their cards with billions of FLOPS, but this number denotes only computational power of the card; often more important readouts such as memory latencies are not presented. The benchmarks are usually targeted at graphics operations for which the GPUs are designed, but our task may have completely different demands.

Having a more complete benchmark suite is problematic. Although the architectures of GPUs are similar and OpenCL provides us with general framework, the internal parameters of each benchmark would have to be tuned for each GPU in the same way as the production code is tuned for peak performance. One setting cannot fit all GPUs.

Articles concerning GPU algorithms use many graphics cards but except in a case we possess exactly the used GPU it is complicated to predict how would their algorithm behave on our GPU and which one is actually better.

Despite all of these problems that we need to be aware of, results of benchmarks allow the reader to estimate the possible impact of using GPU in practical applications. The GPU we used was the state-of-the-art card at the time when this thesis begun and it may be considered as a good representative of high-end GPGPUs.

## 3.2 Execution Time

The exact definition of execution time is very important. The program execution consists of compiling the CL source code[10], sending the data from RAM to GPU global memory, execution of one or more kernels and then gathering the results back to RAM where a common application can use them. The wall time of the program execution is important if the usage of GPU is only casual or the executable file is used within some script. If the GPU is used intensively but only for the specific task alone, both the data-transfer times and execution time are of our interest but the compilation which is performed only during the start-up is not important. When the data are processed primarily on the GPU and do not need to be transferred in large quantities to main memory (or some other I/O device), we care solely about the execution time of the kernel(s) itself.

In our benchmarks of the sorting and set intersection algorithm implementations, we start the stopwatch when the data are about to be transferred to GPU and stop it when all output is gathered back in RAM. This approach was chosen because we consider improving existing applications with GPU usage. As the development of GPU programs is still rather complicated, having the majority of a complex application running on GPU is currently not realistic – that is why we try to find achievable speedup of computationally intensive and well parallelizable operations.

Despite that, knowing the memory transfer times may be beneficial for understanding the behavior of the non-homogenous CPU-GPU system. Therefore, we will mention them in the results of our experiments.

## 3.3 Data Selection

The data we use are synthesized using either standard POSIX rand() function or in case we need unique numbers we use Galois linear feedback shift register [10] initialized with a pseudo-random number obtained using the rand() function.

10 This can be ommited by storing the compiled code. In NVidia case it is assembler, therefore not the final machine code and we have to assume that there is some hidden compilation phase between loading the assembler from persistent memory and actually executing it on the GPU.

We generate this data only once for each combination of parameters. When the benchmark is started it loads these data from disc and runs all executions of all algorithms on these data. Therefore, the results are directly comparable.

For easy replication the most important test data are stored on the enclosed DVD. Test data for large sets and asymmetric sets used in intersection benchmarks are not included due to enormous space requirements and insufficient space on the DVD.

## 3.4 Size of Work-group

Optimal size of work-group may vary for each algorithm. It has been observed that this size is between 128 and 512 work-items. Therefore, we run all benchmarks with work-group size 128, 256 and 512. After the mean value of the execution time is computed we use the best value from these three means. We assume that the optimal value depends on the implementation of algorithm and GPU rather than on the data itself.

There are more options for the work-group size but implementations of our algorithms require that the work-group has power of two work-items. This simplifies some computations in the program, allowing it to run slightly faster.

# 4. Sorting

Sorting is a very important part of many computing tasks. In databases it is used for index creation, query processing, user-requested sorted output as well as removal of duplicities, verifying uniqueness and grouping. In computer graphics the construction of spatial data structures required for rendering and ray-tracing is also basically a sorting process. Structures such as octrees or k-d-trees require sorting, and these can be used in many physical simulations – molecular dynamics, collision detection, particle-based fluid simulation.

There are basically two types of sorting algorithms. Comparison sorts are based on comparison of elements (keys), others split the input set into buckets based on their absolute values – these are called distribution sorts[11]. This thesis is focused on the first class of algorithms.

## 4.1 Related Work

Initially, the GPUs lacked general-purpose programming support although it was still possible to use the graphics primitives to actually do computations. One of the earliest successful implementations of a sort algorithm was bitonicsort by Govindaraju et al. [11], later improved as AbiSort by Greß and Zachmann [12].

Quicksort is generally considered the fastest sorting algorithm on CPUs. GPU implementation with good results was reported by Cederman and Tsigas [13].

Mergesort was implemented by Satish, Harris, and Garland [14], providing a comparison with their radix sort. Radix sort was also implemented by Merrill and Grimshaw [15].

## 4.2 Implementation

All algorithms mentioned below are implemented for both key-value pairs and keys only, with 32-bit floats used as the keys and 32-bit values bound to the keys. Since the code is templated it is possible to use the algorithms with another data-types as well but some hard-coded constants[12] are optimized for 32-bit values.

11 The most famous representative of distribution sort is the radixsort.
12 For example sizes of local memory buffers.

The keys are randomly shuffled within the sequence – we do not consider any optimization based on the assumption that the sequences may be partially sorted. There may be duplicities among the keys but we do not require our algorithms to be stable.

Bitonicsort and mergesort implementations assume that the length of input sequence is a power of two. This simplifies the implementation, but it has probably only marginal effect on the performance. Quicksort internally requires to be able to sort sequence of any length, therefore, there is no such external requirement for the input sequence length.

As the algorithms for sorting using an external memory are well inspected, we assume that the sorted sequence can fit into GPU memory.

### 4.2.1 Quicksort

The idea of quicksort [16] is to select one element from the sorted sequence, and generate two output sequences – one with elements lesser than the selected pivot and one with greater ones. The pivot itself can be then positioned at a place computed from size of the sequence of lesser elements. In the next iteration, all not-yet-sorted sequences are processed until such sequence exists. With a logarithmic number of such iterations (in average case) it has time-complexity $O(n \cdot \log(n))$. In the worst case the number of iterations can be linear with the resulting time-complexity of $O(n^2)$.

The common quicksort algorithm going from both ends of the sequence and swapping elements does not require any additional memory. However, with multiple work-items in a work-group, this approach is not applicable on GPU. Therefore, our algorithm uses an additional buffer of size *n* for the output sequences – in each level of recursion the input and output buffers are interchanged. Another buffer (but smaller) is used for a queue of not-yet-sorted sequences.

There are two levels of parallelism in the algorithm. The first, more obvious, is to process the sequences in parallel as there is no synchronization needed between separate sequences. In the second level of parallelism single sequence can be processed in parallel as well. However, synchronization of the output is required here.

We will describe the sequence parallelism first. A list of not-yet-sorted sequences as pairs of start and end positions is provided to the kernel. A work-group is spawned for each of these sequences (called *jobs* in the source code, therefore, we call this list *job list*). It has an advantage of a simpler communication – if only one work-group processes the sequence, work-items can communicate through local memory and barriers. If the sequence is shorter or equal to the size of the work-group, it is sorted by selection-sort (variation QSSS) or bitonicsort (variation QSBS), otherwise quicksort split is executed.

The selection-sort assigns an element from the sequence to one work-item and then requires all work-items to iterate through the sequence, counting elements lesser[13] than their own. The sequence is located in the local memory and since the work-items are synchronized, the reading results in a broadcast instead of a bank conflict. After the work-items iterate through the whole sequence each work-item knows the position of its element. Bitonicsort is more thoroughly described in the next section.

The quicksort split uses two circular buffers in the local memory, each one sized to the double size of the work-group – one for elements lesser than the pivot and the other for greater elements. Note that the pivot is chosen as the first element in the sequence. All work-items are reading elements from the input sequence in parallel ($i$-th work-item reads element $k \cdot W + i$ where $W$ is the size of the work-group) and after comparing it with the pivot atomically increment buffer position in the appropriate buffer and copy the element there. When the buffer is at least half-full, this half is flushed into the global memory. The memory layout is

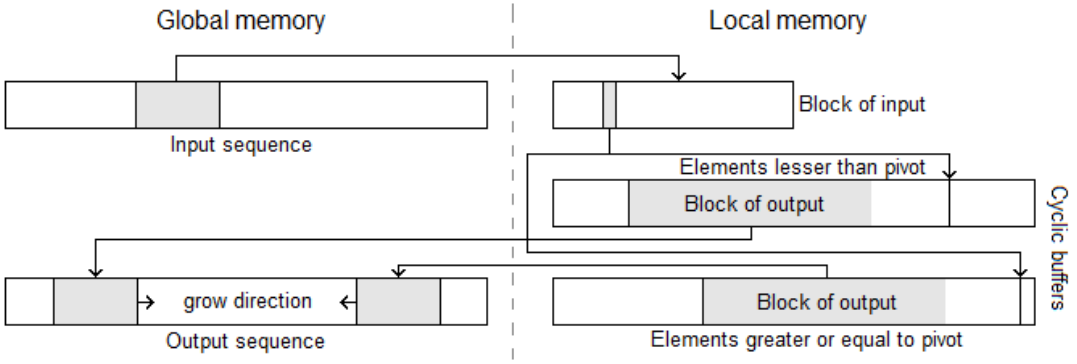



Figure 4.1: Local memory buffers in quicksort implementation

13 The original index of the element is used as the secondary key for comparison. Therefore, the resulting index of the element in the new sequence is not ambiguous.

illustrated in the figure 4.1. As just single work-group splits the sequence, there is no synchronization required.

If there are not enough sequences for each work-group[14], the sequence must be processed by multiple work-groups. These work-groups are called a team. Assigning input elements is simple – $i$-th work-item in $j$-th work-group processes element $k \cdot T + j \cdot W + i$ where $T$ is the number of work-items in the team.

For synchronization of writing into the output sequence, there is a second list passed to the kernel along with the list of not-yet-sorted sequences. This is initially identical to the job list as it contains positions where the sequence should be written to. Similarly as the sequence specification (job) the pair of these positions is shared between work-groups in the team. Since the work-group needs a space to flush its buffers it atomically increments (or decrements in case of the greater elements) the output position and seizes the space in the output sequence.

There is a problem with the pivot itself – it is important to determine which work-group writes it to the output sequence. We have to split the input sequence into three parts: elements lesser than than the pivot, elements greater than the pivot and the pivot itself. If we include the pivot into one of the parts and the other part is empty the requirement to shorten the sequence at least by one element will not be fulfilled. Therefore, the position of the pivot can be computed only after all work-groups have finished. Then it is placed between the two new subsequences and it is not moved anymore. To write the pivot is a task for the last work-group processing its part of the sequence.

The subsequences are constructed iteratively using the two pointers to current end of the subsequences. However, there is no protocol which could determine if these pointers will not be updated just from their monotonic nature as these are not strictly monotonic. This is why yet another list with counters with amount of work-groups in each team is used. After the work-group is finished (the pointers are not going to be updated) this counter is decremented. If it drops to zero this work-group will write the pivot on the last empty position.

14 NVidia GeForce GTX 580 has 16 SM and with the register count and local memory usage each SM is able to handle 3 work-groups of size 256 (or 6 work-groups of size 128) in parallel.

### 4.2.2 Bitonicsort

Bitonicsort [17] is parallel sorting algorithm by design. It requires $O(n \cdot \log^2(n))$ comparisons for sorting that can be processed in $O(\log^2(n))$ parallel steps. Therefore, with *p* processors having theoretical time-complexity

$$O(max(\frac{n}{p}, 1) \cdot \log^2(n))$$

Bitonicsort can compete with other classical $n \cdot \log n$ sorting algorithms if $p \geq \log n$ .

The idea is that the sequence is split into two equal-size subsequences and each is sorted using recursive application of the same algorithm – the first in the ascending order and the second one in the descending order. Then these two sequences are joined using bitonic-merge: each element from the first sequence is compared with the element on the same position in the second sequence and if these two are not properly ordered, they are swapped. After that each sequence is split into two parts and joined again using recursive application of the bitonic-merge. When these two sequences are concatenated they form a sorted sequence.

Order of comparisons is depicted in the figure 4.2 - each white box is one bitonic-merge and each stripped box is one bitonicsort. The arrows show the direction of comparison.

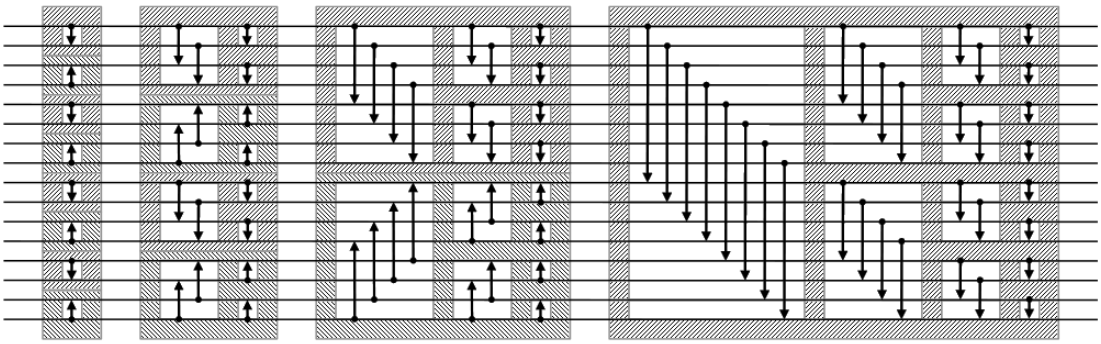


Figure 4.2: Bitonicsort

Let us denote the basic version of the algorithm working directly on global memory by BSB (bitonicsort basic).

As the number of available work-items is generally much lower than the number of elements to sort, each work-item can process multiple elements. This

vectorization technique is usually suggested for ATI cards. This variation will be denoted by BSV (bitonicsort vectorized).

Another improvement is to use the local memory instead of accessing the global memory with a high latency. You can see that multiple bitonic-merges may operate on the same small part of the memory. Usually, each bitonic-merge is processed in a separate kernel. Nevertheless, the bitonic-merge on short sequences can be completely processed within a single kernel, loading the data into the local memory at the beginning and writing them at the end, while removing many unnecessary loads and writes. In our case, these short sequences can be up to $4 \cdot W$ in length ($W$ is the work-group size) because of the available size of the local memory. We call this variation 'bitonicsort using local memory' – BSL.

### 4.2.3 Mergesort

Mergesort [18] was a popular sorting algorithm especially in the times when RAM memory was expensive – it can be implemented using cheaper external memory such as tape drives. However, compared to the quicksort the in-place implementation is more complicated and, in practice, it is often outperformed by quicksort.

In the basic serial implementation, the input sequence is split into two halves and both are sorted using recursive application of the mergesort algorithm. Then these two sorted sequences are iterated through moving the smallest element from both sequences (which can be found only at the beginning of either sequence) into the output sequence. This implementation has both the average and the worst time-complexity $O(n \cdot \log(n))$.

There are multiple ways how to parallelize the mergesort algorithm. The first comes from its divide-and-conquer nature: sorting of both subsequences can be executed in parallel. However, the linear merge phase is strictly sequential and can be found to be a bottleneck.

This merge can be parallelized as well. Let us define the greatest lower bound (GLB) of $x$ in a sorted sequence $S$ as the highest index of any element of $S$ lesser or equal to $x$. The parallel merge then consists of the following steps where steps 2 and 3 are processed in parallel:

1. GLB $l$ of the element at position $m = n/2 - 1$ in the first subsequence is searched in the second subsequence (e.g. using binary search).
2. The first half of the first subsequence is merged with the first part of (with regard to the GLB $l$) the second subsequence.
3. Similarly, the second half of the first subsequence is merged with the second part of the second subsequence.

With enough processors this leads to a logarithmic work on each of the $\log(n)$ recursion levels giving the time-complexity of $O(\log^2(n))$. With a limited number of processors $p$ the sorting takes $O(n/p \cdot \log(n/p))$ time to sort the subsequences in parallel and then $O(\log(n) \cdot \log(p) + n/p)$ in the parallel merge phase. Therefore, it has a total time complexity of $O(n/p \cdot (\log(n/p) + 1) + \log(n) \cdot \log(p))$.

This parallelization approach can be used on CPUs but it is not suited for cooperative threads on GPU. The communication between the parallel threads would be complicated – the GLB search would have to share the results or all processors would have to search the GLB individually.

```
parallel_merge(S1, S2) {
    if (length(S1) == 0) {
        return S2;
    } else if (length(S2) == 0) {
        return S1;
    } else if (length(S1) < MIN && length(S2) < MIN) {
        return sequential_merge(S1, S2)
    } else if (length(S1) > length(S2)) {
        return parallel_merge(S2, S1)
    } else {
        m := length(S1)/2 - 1
        l := greatest lower bound of S1[m] in S2
        parallel {
            T1 := parallel_merge(S1[0 .. m], S2[0 .. l])
            T1 := parallel_merge(S1[m + 1 .. length(S1) - 1],
                                 S1[l + 1 .. length(S2)])
        }
        return concatenate(T1, T1)
    }
}
```

Code listing 4.1: Parallel merge appropriate for CPU

A different approach was used for the GPU implementation. The algorithm is divided into logarithmic number of phases as the levels of recursion in the classical mergesort. After each phase, the length of sorted subsequences is doubled and the number of subsequences is halved. In each phase, one element is assigned to each work-item, the work-item computes the position of this element in the new subsequence and writes the element on this position.

When two sequences are merged the new position of an element from the second sequence is equal to the sum of its current position within the second sequence and the GLB of this element in the first sequence. For an element *x* in the first sequence it is, likewise, the sum of its current position within the first sequence and the greatest index of an element lesser than *x* in the second sequence (let us call this GLT, 'greatest lesser than...'). Notice that the relation here is strict, unlike in the GLB case.

As both sequences are sorted the GLB and GLT can be found using some sorted array search algorithm. In our implementation, either by binary search or by interpolation search. These are more thoroughly described in Section 5.2.1. There are only minor modifications – those algorithms are designed for exact match search. Since we need to search GLB or GLT, a different compare sign is used in the binary search or in the interpolation search implementation.

This algorithm scales even better than the aforementioned one (rather suitable for CPU than GPU). Each element in the sequence can be processed in parallel with possible 100% utilization of the processors (if *n* is a multiple of *p*). There is also no need to synchronize the work-items.

Time-complexity is $O(n/p \cdot \log(n) \cdot S(n))$ where *S(n)* is the time-complexity of the search function. In case of the binary search, it is $O(n/p \cdot \log^2(n))$ in both the average and the worst-case. In case of the interpolation search the average time-complexity is $O(n/p \cdot \log(n) \cdot \log(\log(n)))$ but the worst-case is $O(n^2/p \cdot \log(n))$.

The first of the sequences is read in a coalesced way but search algorithms have usually worse memory pattern.

For simplicity reasons the first implementation (mergesort basic – MSB) used selection sort for sorting the subsequences with length $4 \cdot W$ and each pair of merged

sequences was processed in a separate work-group. This simple approach causes unnecessary restraint of parallelization. Therefore, the second implementation (mergesort in teams – MST) allows more work-groups to merge single pair of sequences. The code might look more complicated but the idea prevails.

In the next implementation, the selection sort was substituted by the bitonicsort in the local memory (called mergesort combined with bitonicsort – MSCBS). The fourth implementation executes bitonicsort only on sequences of length 32, after that mergesort with the binary search in local memory is used (MSCBS32).

The last two implementations are variations of MSCBS and MSCBS32 with the binary search substituted by the interpolation search – we denote them by MISCBS and MISCBS32.

## 4.3 Results

In this section the results of our benchmarks will be presented. At first, the variations of each strategy will be shown. Then we will examine the comparison of CPU sorts and the best representatives from each GPU sort strategy. The exact numeric results (execution times) are located in Appendix A.

### 4.3.1 Quicksort

Quicksort shows very similar results for both variants of short-sequences sort, moreover, the results do not differ even between the keys-only version and version with 32-bit values. This is why we present only the chart of the version including the 32-bit values.

We can state that the sorting method for short sequences is not significant for the total results, the execution times are fully within the deviation intervals.

Both implementations have remarkably high deviations and the sorting time depends highly on the particular set we are sorting. It is probably caused by the amount of communication between work-groups – different order of atomic operations may change the order of jobs in job lists. That is also why we can see notably different rates for sets with varying sizes – these are different sets, while for one size we always use the same set for all measurements.

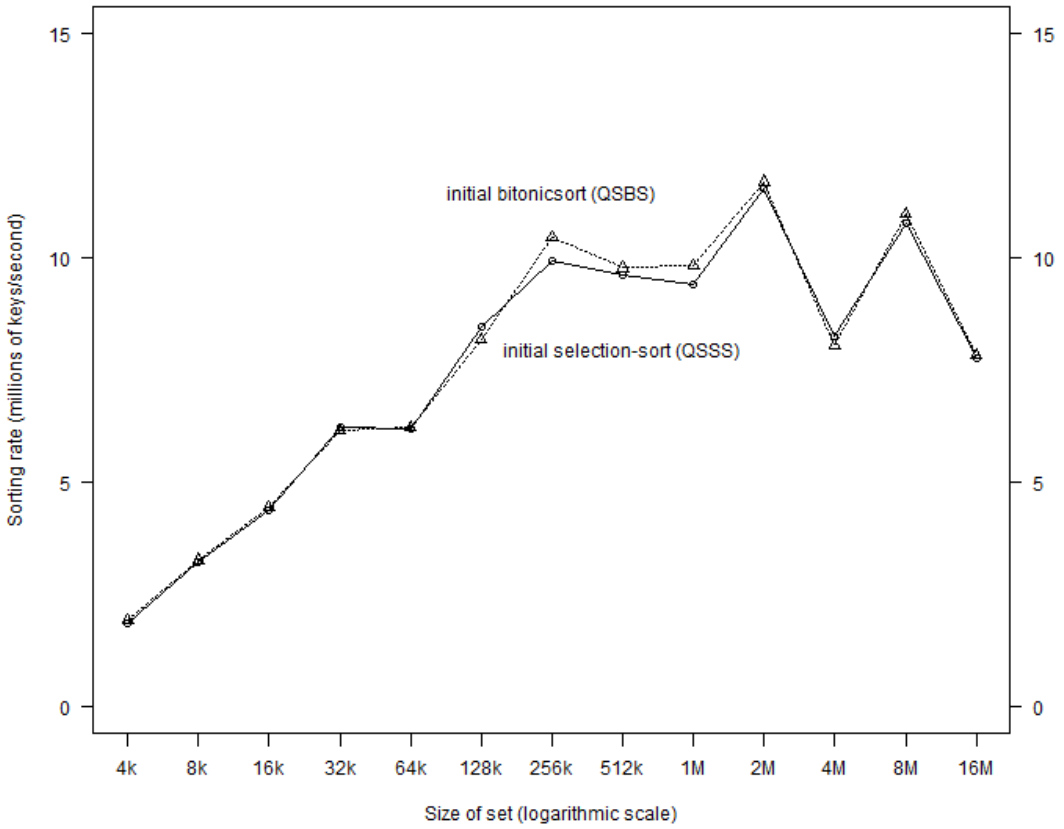



Chart 4.1: Quicksort variants with 32-bit values

### 4.3.2 Bitonicsort

Two optimizations of the basic bitonicsort were implemented but the vectorization in BSV cannot be interpreted much like an optimization – it has even worse performance than the basic version (BSB). The optimization did not help us probably because there is only minimal amount of the computation between two loads, not allowing any kind of overlapping of computation and data loading.

On the other hand, using the local memory in the variation BSL is a great improvement, boosting the performance by approximatly 28% for the keys-only version and 34% for the version with key-value pairs.

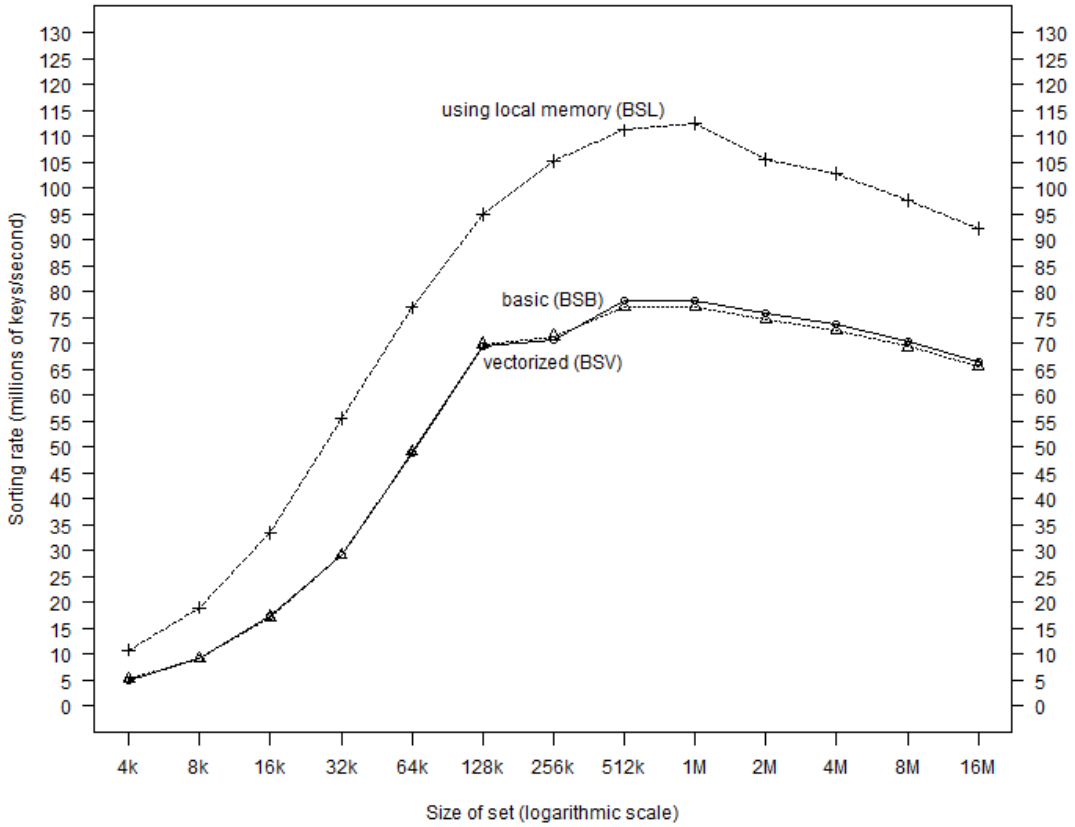



Chart 4.2: Bitonicsort variants without values

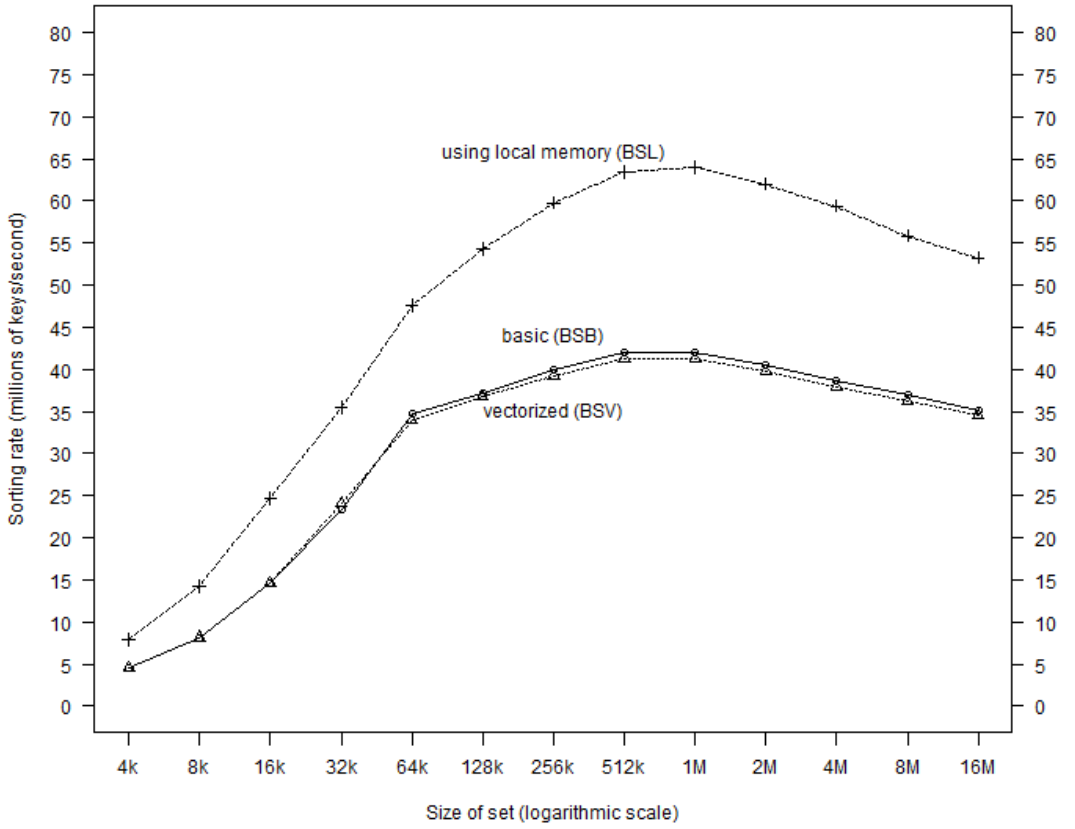



Chart 4.3: Bitonicsort variants with 32-bit values

### 4.3.3 Mergesort

The first implementation (MSB) with one work-group per sorted sequence does not perform very well as the parallelization options are limited for a long period of time. This can be also observed from the almost constant sorting rate from sequences of size 128k whereas the sorting rate of all other strategies still grows.

The versions with initial bitonic- (MSCB and MISCB) and combined bitonic- and mergesorts (MSCB32 and MISCB32) have similar performance although the bitonic-only version is a bit faster. From about 512k sequences, the difference between smart initial sorting and the selection sort is almost constant, which corresponds to the fact that only constant number of sorting levels (phases) differ – the change is not proportional to the size of total input.

On the uniformly distributed set of values, the variations with interpolation search (MISCB and MISCB32) gain over the binary search. This is significant mostly for sequences consisting of more than 1M keys.

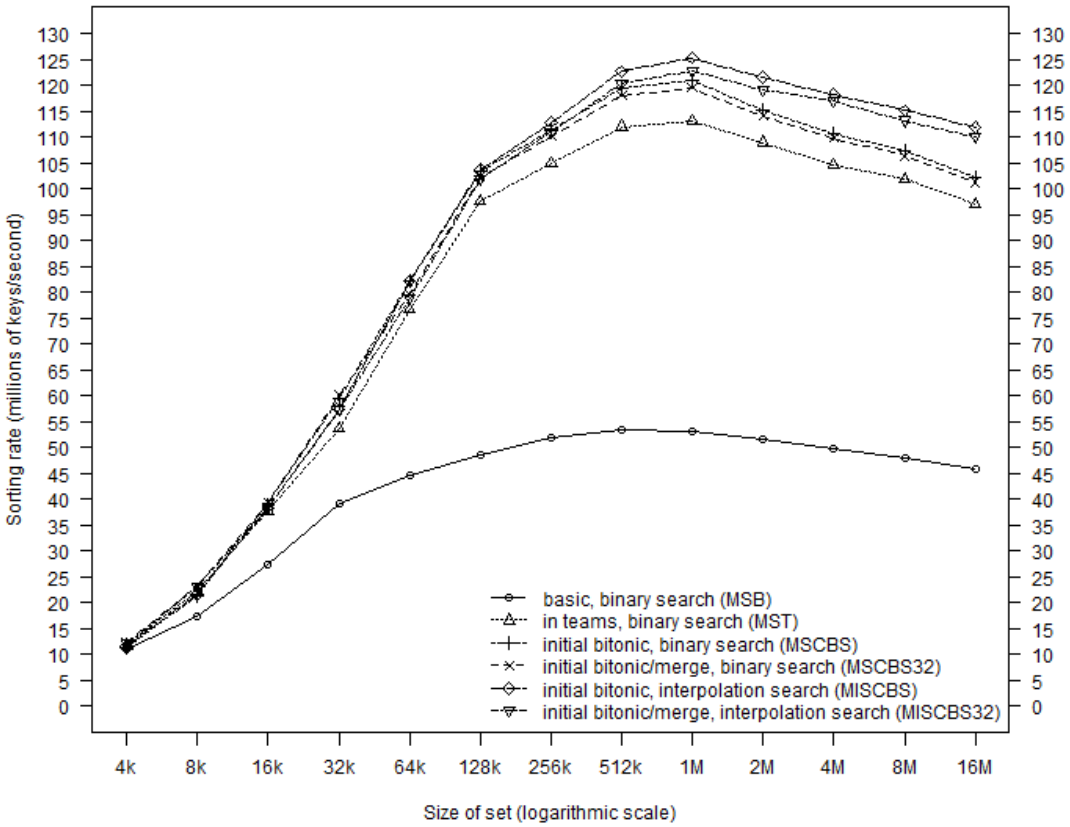



Chart 4.4: Mergesort variants without values

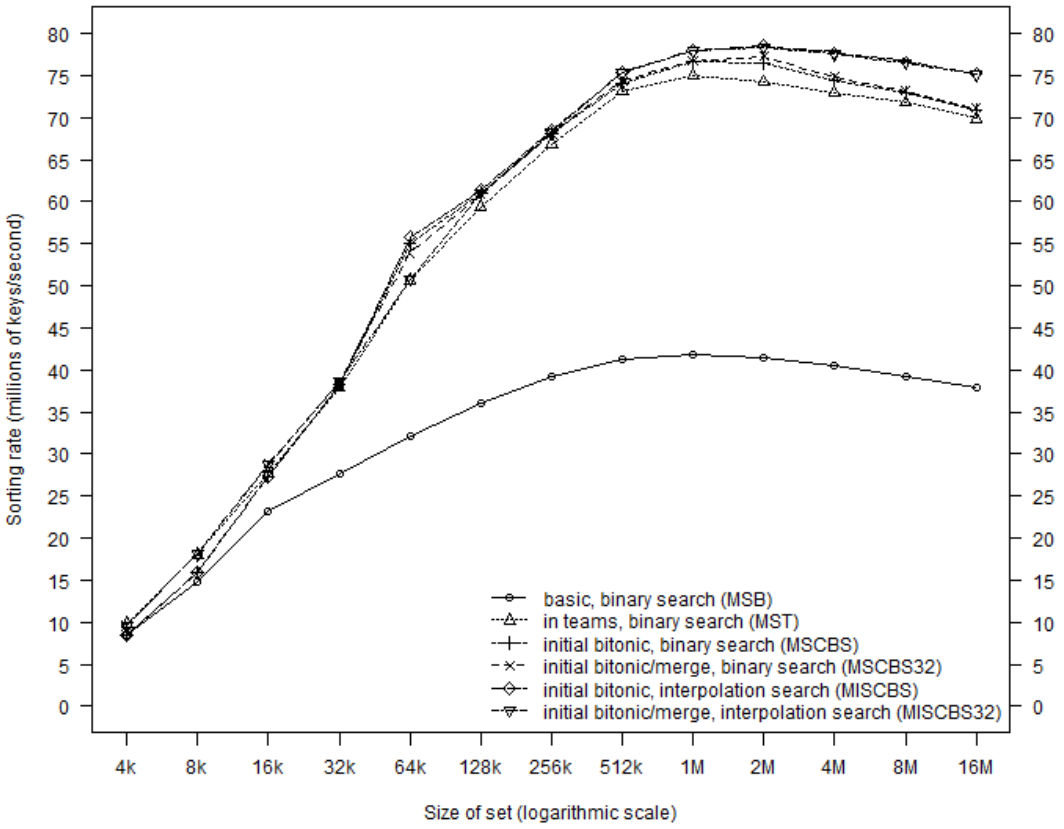



Chart 4.5: Mergesort variants with 32-bit values

### 4.3.4 Comparison of CPU and GPU Based Sorts

We have chosen two representatives of CPU sorts. The first is std::sort from the STL library; it is not optimized for top performance but it is a good standard for future the comparison with other highly optimized CPU sorts. This sort algorithm runs in a single-threaded apartment.

Second is tbb::parallel_sort from Threading Building Blocks library [19]. It uses multiple worker threads for sorting. Therefore, it can use all CPUs available.

Quicksort does not seem like the best choice for GPUs as it can hardly compete even with the std::sort. There may be various reasons: the algorithm is rather complicated and uses many registers. This causes a problem with GPU occupancy as there cannot be the maximum number of work-items scheduled at one time because they would require more registers than the hardware can provide. Another problem is using atomic instructions. Although the global memory atomics are mostly used once per work-group, there is the need to synchronize the work-groups. Local memory atomics are also widely used. Moreover, there is a non-trivial cooperation with CPU where the host program selects some parameters according to the results of the last phase. Therefore, CPU and GPU sometimes have to wait for each other.

Mergesort and bitonicsort offer better results. Both can compete with std::sort on sequences of 8k keys and are significantly faster on longer ones. They are even faster than the tbb::parallel_sort on sequences longer than 32k keys.

As all presented sorting algorithms are super-linear[15], the CPU algorithms have the best sorting rate on shortest sequences, then the performance slowly decreases. Due to GPU latencies, the peak performance of GPU strategies is between 2M – 4M sequences for mergesort and 1M – 2M for the bitonicsort. Then we can also see the decrease caused by the super-linear nature of sorting algorithms. The decrease is even faster than in the case of CPU sorts. This is caused by worse theoretical time-complexity of the parallel algorithms: $O(n/p \cdot \log(n) \cdot \log(\log(n)))$ for MISCB and $O(max(n/p, 1) \cdot \log^2(n))$ for BSL, compared to the $O(n \cdot \log(n))$ used in CPU sorts. The number of processing units remains so the sorting rate decrease is faster.

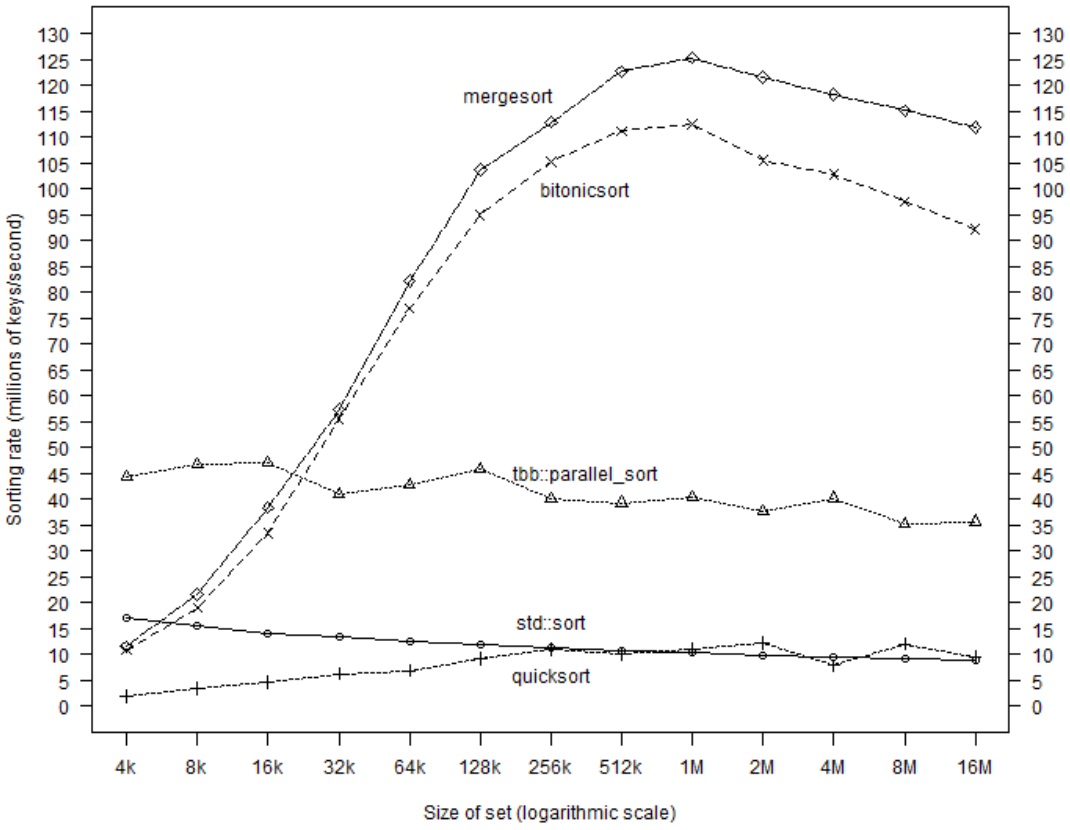



Chart 4.6: CPU vs. GPU strategies - without values

15 Considering that the amount of elements in sorted sets is much higher than the number of processing units.

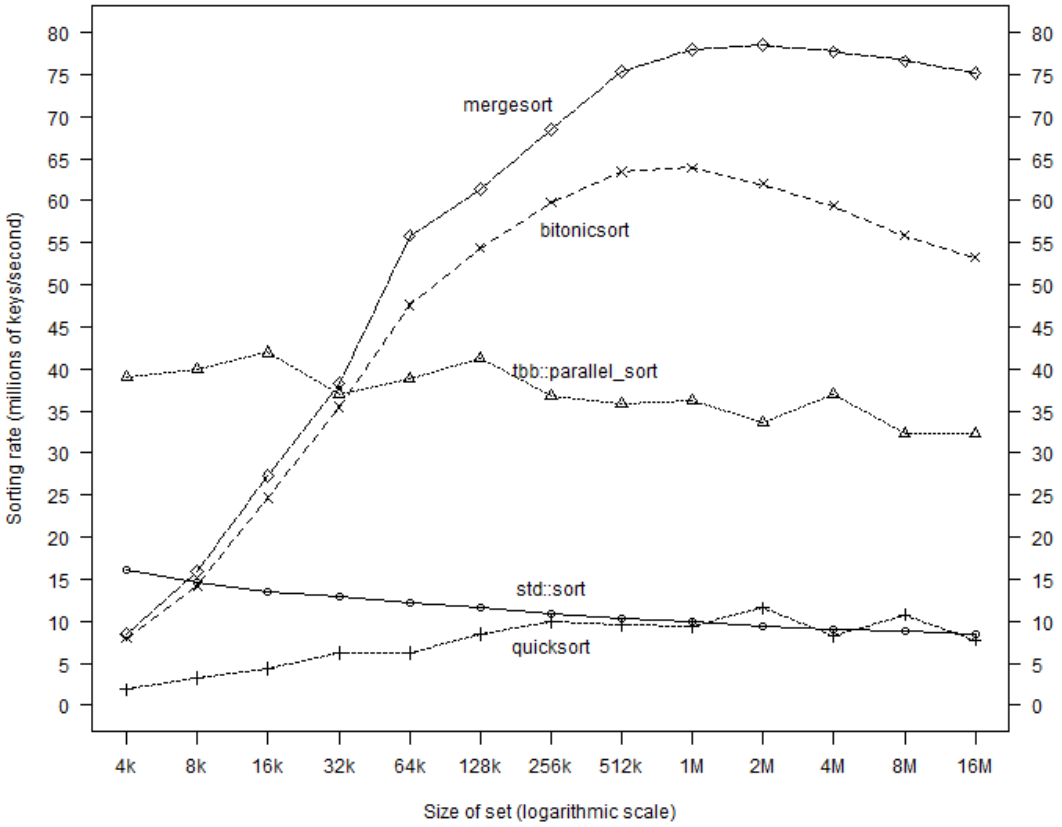



Chart 4.7: CPU vs. GPU strategies - 32-bit values

As a result, we can say that the mergesort on GPU can be up to 3.1 times faster than tbb::parallel_sort and 12.4 times faster than std::sort on data without values and 2.4 times or 9 times faster respectively on the data with 32-bit values. As the memory moves on GPU are expensive we can assume that larger values would render smaller speedup on the GPU. For exact numeric results we refer to Appendix A.

We note that the host-GPU-host memory transfer times are far from being negligible. Our results show that these take about 50 – 60% of the total computation time in case of key-value pairs and 35 – 50 % in the keys-only case.

## 4.4 Future Work

In this section we present some ideas for future improvements.

First, the set may be already partially sorted. If we were able to detect these sorted subsequences (or get them along the data) we could adapt our mergesort implementation to skip several levels of sorting. This would require to pass a list of these subsequences in the same way as we used jobs in our quicksort implementation. With different sizes of the sorted sequences some more load-balancing would be required. Unlikely to the quicksort, the work-groups would

not need to synchronize accesses to any parts of the memory because only the CPU would decide about new jobs, knowing which sequences are sorted in each phase of the algorithm.

More complicated algorithms usually require more registers and carry some overhead. The only option to determine the overall performance is an implementation and an experiment.

Another variation would be the insertion of a small amount of not-sorted data into larger sorted set. This is a common case when new rows are inserted into an indexed database table. It is not much complicated with sequential access, in fact, it is a merge operation. Transfer of the large sorted set to GPU would probably be too expensive but if the large set was already present in the GPU memory, the comparison of sequential merge and parallel one might be interesting.

# 5. Intersection

We have defined the problem of intersection as a task to find a common subset of two sets of 32-bit numbers. The numbers in each set are unique and uniformly distributed[16] across the 32-bit universum.

There are several basic ways how this problem is solved in serial algorithms. The simplest approach is the non-indexed nested-loop join (NINLJ) testing each pair from the cartesian product of the two sets. This imposes no requirement on the data but as the algorithm has quadratic time-complexity, it is not practical for large sets.

Indexed nested-loop join (INLJ) requires representation of one of the sets in a search structure such as sorted array or tree. The other set is iterated through and query into the structure is performed for each element.

Sorted merge-join (SMJ) sorts both sets and then selects the common elements in a single pass through both sets. This algorithm is basically a variation of mergesort. INLJ and SMJ algorithms will be discussed in section 5.2.

Hash-join (HJ) is very similar to INLJ but it uses hash-map as the search structure. Naturally, the hash-map has to be constructed prior to the queries. Here we also need a good hash function distributing the input data uniformly accross the hash-table. As we assume that the input sets itself are distributed uniformly accross the universum, simple modulo should be sufficient. Hash-joins are the main topic of section 5.3.

From the implementation perspective, the absolute size of the sets are an important property. Those algorithms presented in sections 5.2 and 5.3 assume that there is enough memory to load and process both sets inside the GPU memory; section 5.4 studies how to circumvent this requirement.

## 5.1 Related Work

Although geometrical intersection is a problem often solved on GPU, the set intersection on GPU is not as commonly studied. Resen and Pagh [20] suggest compressed bitmap structure called BatMap similar to Cuckoo hash-table recommended by Alcantara et. al. [21]. Bingsheng et. al. [22] explores both parallel

16 In fact we have used pseudo-randomly initialized Galois linear feedback shift register [10].

merge-join exploiting the local memory and variation of hash-join splitting the sets using radix parititioning. Wu et. al. [23] uses INLJ for intersecting a short and long sets. Ao et. al. [24] also uses INLJ and suggests precomputing probable position of elements using linear regression.

## 5.2 Intersection of Sorted Sets

As sorting algorithms on GPU were described in section 4, here we focus only on the second phase, assuming that one or both sets are initially sorted.

Although the single-pass algorithm is apparently optimal with serial hardware, doing this on GPU would be a great wasting of resources.

Our options to parallelize the task are very similar to those with the merge operation in the mergesort algorithm. We can either split the two sets to pairs of subsets, which can then be sequentially processed (SMJ), or search for elements from first set in the second set individually (INLJ). In fact, the second approach requires only one set to be sorted, although having both sets sorted may yield a better performance (see below).

With both sets sorted, it would not be difficult to get the result sorted as well. It would require filling a part of the memory sized equally to the original set with zeroes and then copying the found elements on their original positions. After that we would use a compact non-zero elements operation, which is a variation of well-known parallel scan [25]. Special handling of zero would not pose a problem. Nevertheless, as our problem description does not require outputting sorted sets this feature was not implemented.

### 5.2.1 Search Algorithms

In these search algorithms, each work-item is assigned one number in the first set, and then it tries to find this number in the second set. It does so by maintaining an interval from the second set (initially encompassing the whole set) where the searched element can be found. In each iteration it selects a new index from the interval and probes the number found in the set on this index. The probed element (pivot) is compared[17] against the searched one, and according to the result the

17 As there is neither a three-way compare operator in ISO C99 nor OpenCL intrinsic for the three-way compare the actual compare is done twice, however as the pivot is loaded into registers there is only a minimal overhead.

number is either found, or the left or right subinterval is selected as the interval for next iteration. If the interval becomes empty, the algorithm ends – the number is not found. The search is summarized in code listing 5.1.

The initial loading of numbers from the global memory, where the first set resides, is coalesced, therefore, we can expect good performance for this part. However, the lookups in the second set may be considered random, causing a lot of memory transactions. Having a separate memory transaction for each work-item in each phase is the worst-case scenario. Nevertheless, with both sets sorted the situation may not be as harsh – if the elements in second set (or the closest ones, in case the second set does not contain them) are near, what could be expected with a uniform distribution, there is a good chance that the memory access will break into only few transactions, unlike single transaction for each work-item in the worst-case scenario.

Moreover, the global memory cache may also help as we are probing only a decent area of memory in the several latest look-ups.

```
searchKey := element from first set
leftIndex := 0
rightIndex := size of second set - 1
while (rightIndex >= leftIndex) {
      pivotIndex := select_pivot()
      pivotKey = element from second set at position pivotIndex
      if (pivotKey == searchedKey) {
            add searchedKey to the set of common numbers
            end
      } else if (keyA > keyB) {
            leftIndex := pivotIndex + 1
      } else {
            rightIndex := pivotIndex - 1
      }
}
```

Code listing 5.1: General search algorithm

## Binary Search (BSS)

This is the simplest method of individual search. The select_pivot function selects the middle point from the interval, letting the search algorithm to halve the interval in each iteration. From the theoretical perspective, the binary search has both estimated and worst-case execution time $O(\log n)$ for each element in the first set, therefore, $O(n/p \cdot \log n)$ for the whole set with $n$ elements, using $p$ processing units (work-items).

### Interpolation Search (ISS)

The idea of interpolation search is very similar to the binary search algorithm, but it exploits the fact that the numbers in a sorted array may form almost linear sequences. The pivot is selected according to this formula:

$$pivotIndex = \frac{searchedKey - leftKey}{rightKey - leftKey}(rightIndex - leftIndex)$$

where *rightIndex* and *leftIndex* define the current range, *leftKey* is the number positioned at *leftIndex* and *rightKey* is the number positioned at *rightIndex*.

This function causes the search algorithm to have estimated execution time $O(\log\log n)$ [26] which is better than in the binary search algorithm. However, the worst-case execution time is $O(n)$.

This algorithm in its base form requires three lookups to the global memory in each iteration: one for the pivot itself and two for keys on the left and right side of the interval. One lookup can be spared by cacheing the keys on interval sides (in each iteration only single side of the interval changes) but as the changing side is set to the place *next* to the pivot, the element should be loaded. If this is not suitable, the key can be approximated – in our implementation it is set just to the value of pivot itself. The values of the first and the last element in the set are approximated as well, to be the minimum (0) and maximum ($2^{32} - 1$) of the universum.

### Generalized Quadratic Search (GQSS)

The idea of this algorithm is to combine the advantages of binary search (worst-case execution time $O(\log n)$) and interpolation search (estimated execution time $O(\log\log n)$).

We do so by partitioning the iterations into several phases. In the beginning of each phase, the pivot is selected using single interpolation query. The rest of this phase comprises of alternated binary and unary search queries. The unary search steps have length equal to the square root of size of the interval at the beginning of the phase. The phase ends when the unary search has to change direction because it detects that the searched key has been skipped. This is summarized in code listing 5.2.

```
if (phaseStart) {
    pivotIndex := select_pivot_by_interpolation()
    pivotKey := element from second set at position pivotIndex
    phaseStart := false
    parity := true        // start with unary search
    length := rightIndex - leftIndex + 1
    up := pivotKey < searchedKey
} else if (parity) {
    if (up) {
        pivotIndex := leftIndex + square root of length
    } else {
        pivotIndex := rightIndex – square root of length
    }
    pivotKey := element from second set at position pivotIndex
    if ((pivotKey < searchedKey) xor up) {
        phaseStart := true
    } else {
        parity := false   // continue with binary search
    }
} else {
    pivotIndex = select_pivot_by_binary()
    pivotKey := element from second set at position pivotIndex
    parity := true        // continue with unary search
}
```

Code listing 5.2: Pivot selection in generalized quadratic search algorithm

As the interval is halved during at most three iterations the worst-case execution time is $O(\log n)$. It can be proven that each phase takes constant time in average [27] and as the number of phases is smaller than the number of queries in interpolation search (each phase begins with interpolation query), the estimated execution time is $O(\log \log n)$.

Although this method has the best theoretical background, the computation is more complicated than in the previous methods. We will see how this will exhibit in the benchmarks.

## Initial Lookup Optimization

We have tried to spare the first few lookups using the faster local memory. Each work-item in the work-group loads a single element from the set with equal distances between these elements, and the search algorithm is executed on the array of the fast local memory at first. The second phase continues as usual but with the interval initially set to the one obtained in the first phase.

This optimization was implemented for both binary and interpolation search.

### 5.2.2 Parallel Single-pass Algorithms

Unlike the search algorithms, this algorithm requires both sets to be sorted on the input. The well known sequential single-pass algorithm can be parallelized by dividing the input sets into separate subsets. However, the subsets from the first set must pair to those from the second set – numbers from each subset of the first set should be present only in the pairing subset of second set respectively. This condition guarantees that no number common to both sets may be missed when processing the pairs of subsets in parallel.

There are more problems related to the task – we should balance the sizes of sets to be approximately equal because the execution time of the whole algorithm is dependent on the execution time of the longest subtask (we are assuming that the execution time of the single-pass is linear to the sum of sizes of subsets in the pair).

Another problem is the decomposition granularity. Should the sequential pass be performed by whole work-group, single work-item or rather by single warp?

#### Dividing The Sets

In order to fit the subsets into the local memory and simplify the algorithm, the size of subsets was limited to *N* where *N* is the size of the work-group. Using binary search[18], greatest lower bound for each *N*-th element from first set is found in the second set. Then the multiples of *N* and GLBs in each set are iterated, pairing the intervals from each set. See figure 5.1 as an example – here are the two sets of size *4N* divided into 8 pairs of intervals, each interval having at most *N* elements. These pairs of intervals can be then processed in the local memory (except for the first pair which has one of the intervals empty).

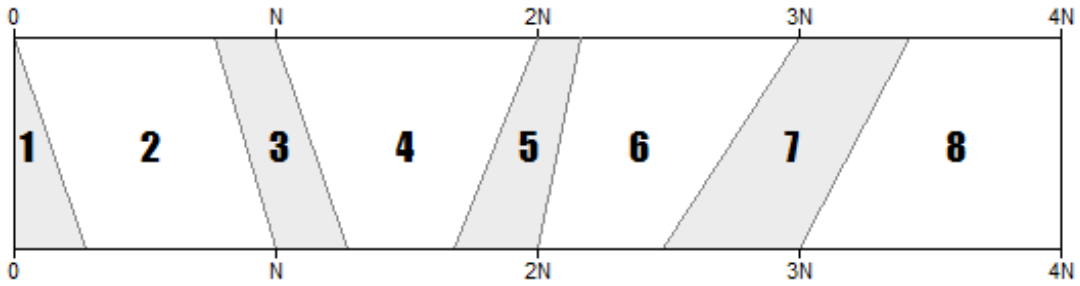


Figure 5.1: Dividing the sets

18 Although interpolation-search proved faster in the global search algorithm, due to some anomalies in floating-point operations it was significantly slower in this case – therefore binary search was used.

The algorithm is summarized in code listing 5.3. As it sequentially iterates through the data it is not parallel by nature. The algorithm was implemented both on CPU ('local search strategy with jobs created on CPU' – LSJCCPUS) and GPU ('local search strategy with jobs created on GPU' - LSJCGPUS). On CPU it needs to move the GLBs from GPU to CPU, wait until the set is processed and then move the interval ranges back to GPU. On GPU single work-item iterates through the data, other work-items in work-group only help with moving the data between the local and the global memory.

```
iteratorA := 1; iteratorB := 1;
if (glbsA[0] != 0) {
    intervalEndA[0] := 0;
    intervalEndB[0] := glbsA[0];
} else if (glbsB[0] != 0) {
    intervalEndA[0] := glbsB[0];
    intervalEndB[0] := 0;
} else {
    intervalEndA[0] := 0;
    intervalEndB[0] := 0;
}
intervalIterator = 1;
while (iteratorA < glbsA or iteratorB < glbsB) {
    if (glbsB[iteratorB] < nextMultipleA) {
        intervalEndA[intervalsIterator] := glbsB[iteratorB];
        iteratorB := iteratorB + 1;
    } else {
        intervalEndA[intervalsIterator] := nextMultipleA;
        nextMultipleA := nextMultipleA + N;
    }
    if (glbsA[iteratorA] < nextMultipleB) {
        intervalEndB[intervalsIterator] := glbsA[iteratorA];
        iteratorA := iteratorA + 1;
    } else {
        intervalEndB[intervalsIterator] := nextMultipleB;
        nextMultipleB := nextMultipleB + N;
    }
    intervalIterator := intervalIterator + 1;
}
```

Code listing 5.3: Dividing pair of sorted sets into pairs of intervals

In fact it is possible to parallelize the loop with a similar technique as those used in the parallel mergesort, however, that is not our case. This approach would be too complicated for the amount of data that is processed.

In our implementation two subsequent pairs of intervals with both sums of interval sizes lesser than *N* can be joined together. This detail is not covered in the code listing 5.3 above.

### Searching for Common Elements

Although having one work-item for each interval and sequentially iterating through the intervals might be the fastest solution from the theoretical perspective (with time complexity $O(n/p)$), this would lead to a non-coalesced memory reading pattern, which is obviously very slow – there would be large stride between the reads. Therefore, one interval is processed by the whole work-group. One work-item is assigned to each element from first interval, searching for equal element in the second interval. This leads to time complexity[19] $O(n \cdot \log(N)/p)$.

## 5.2.3 Results

In this section we will compare the strategies with each other. After that, as no advanced CPU join algorithm for sorted sets was implemented in this thesis, we will use simple sequential merge-join as the counterpart for CPU – GPU comparison.

The sets were sized from $2^{12}$ = 4k to $2^{24}$ = 16M keys. We have used configurations with varying number of common elements – no common elements, 0.1%, 10%, 50%, 90% and identical sets (differently shuffled, of course). As having chart for every configuration would require too much space, we show only those results showing important properties of the strategies. Rest of the charts is provided on the enclosed CD.

### GPU Strategies Comparison

Absolute values show similar performance for all search methods. This is caused by the long-lasting transfer of sorted sets from RAM to GPU and also the transfer of the resulting intersection back from GPU to RAM in all strategies. Therefore, we will analyze the strategies from the next chart where the intersection times are scaled to

19 As the *N* is bound to constant size of the work-group rather than to size of the input, the *log(N)* can be also considered as constant.

the sum of memory transfer times. Since this chart shows relative time of the intersection instead of sorting rate as the other charts, the lower ratio is better.

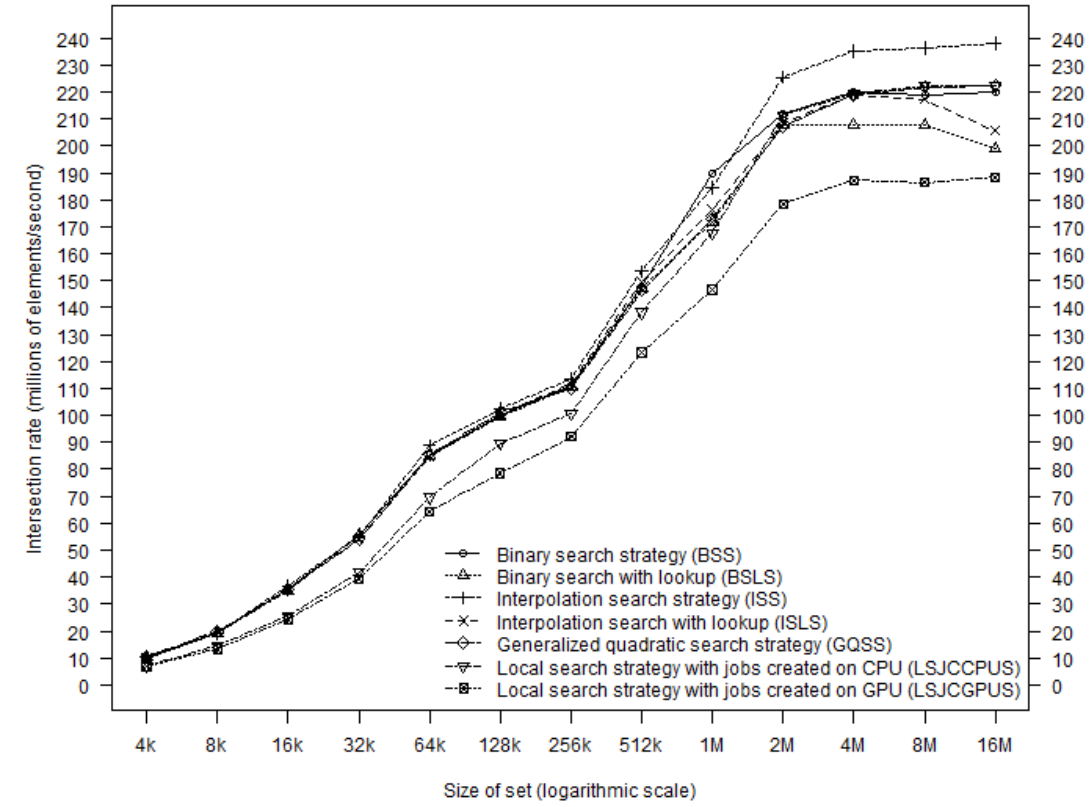



Chart 5.1: GPU strategies on equally sized sorted sets with 10% common elements (absolute)

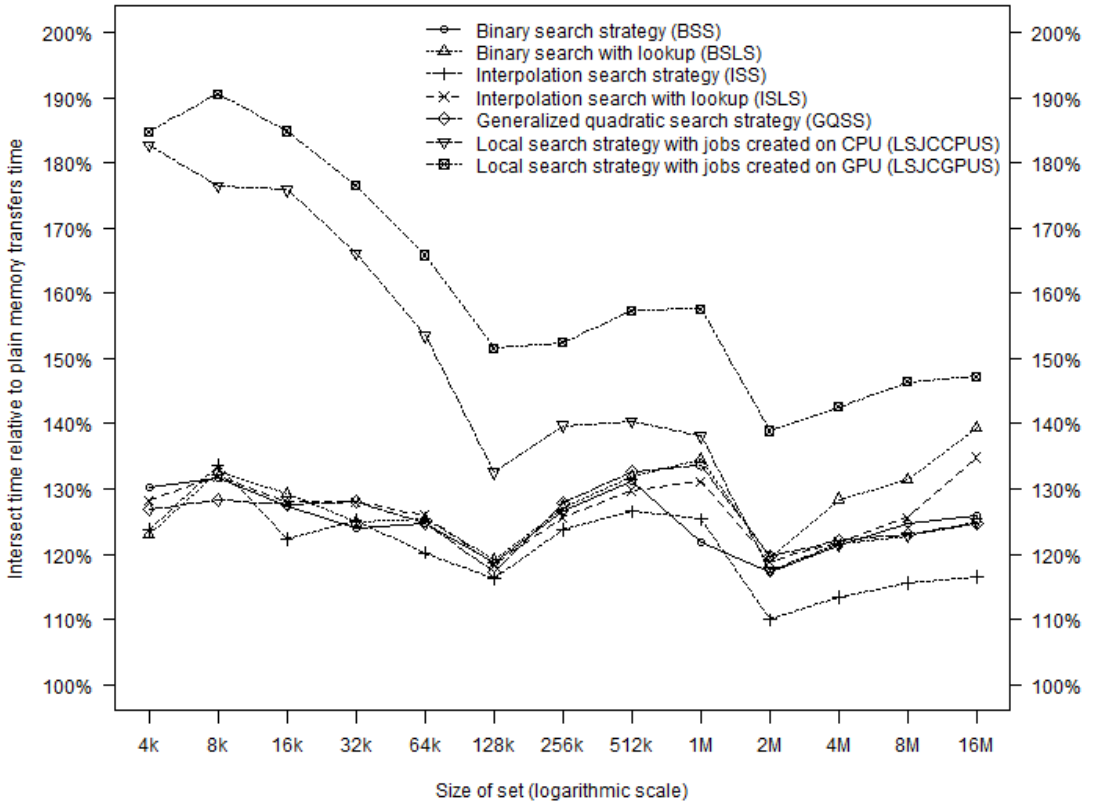



Chart 5.2: GPU strategies on equally sized sorted sets with 10% common elements (relative)

The interpolation search (ISS) is fastest through all set sizes. The look-up optimization (BSLS and ISLS) has not proven useful – in fact, the performance of these variations is worse compared to the basic strategies. Binary search (BSS) can compete with other strategies on smaller sets but for larger ones, it is slower.

Although conceptually very different the generalized quadratic search (GQSS) performs similarly to the local search strategy with jobs created on CPU (LSJCCPUS). Both these offer mediocre performance compared to interpolation search. Local search strategy computed on GPU solely (LSJCGPUS) is the slowest strategy because the serial part is not fitting well to the GPU architecture.

### Asymmetric Sets

All results above come with symmetric set sizes – both sets have the same size. The results below compare different distributions of sizes with the same sum of sizes of the sets. We have used ratios in the form $1:(2^i - 1)$: 1:1, 1:3, 1:7 and so forth. The amount of common elements was set to 4k which is the size of the smallest set used for these benchmarks. This was motivated by the effort for keeping constant memory transfer times for all pairs of sets with equal sum of sizes.

In the charts below, execution times are normalized to the execution time of pair with equally sized sets, marked as empty circle. If the triangle points upwards the first set passed to the algorithm is greater, if it points down the second set is greater. The darker the triangle is the greater is the difference between the sizes of the two sets.

On all strategies extreme values offer the shortest execution times and the higher ratio renders shorter execution time.

The $n \cdot \log(N-n)$ curve expected for binary search has theoretical maximum (the longest execution time) for ratios between 1:1 and 1:3 with smaller first set; this is similar also for the interpolation search. Our results show longest execution time with ratios between 1:7 and 1:15 with smaller second set, for both variants of binary search, interpolation search and for generalized quadratic search as well. With these strategies, the ratio with smaller first set is almost always faster than the other variant.

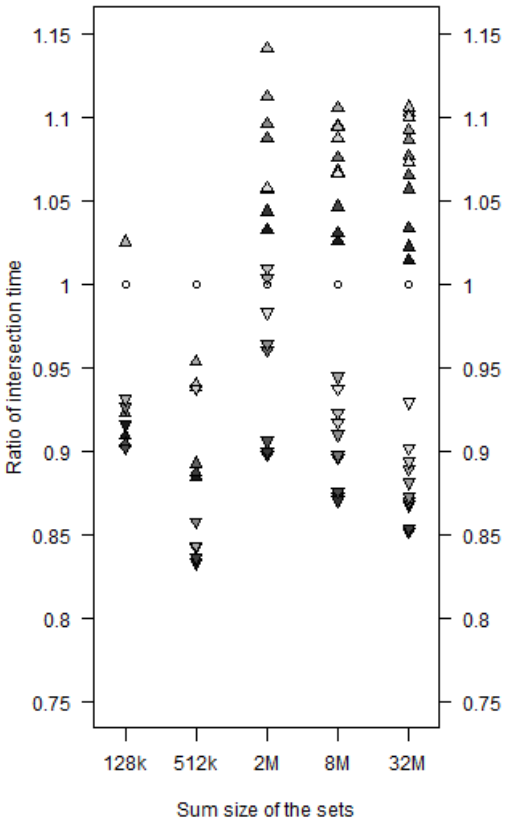



Chart 5.3: Binary search

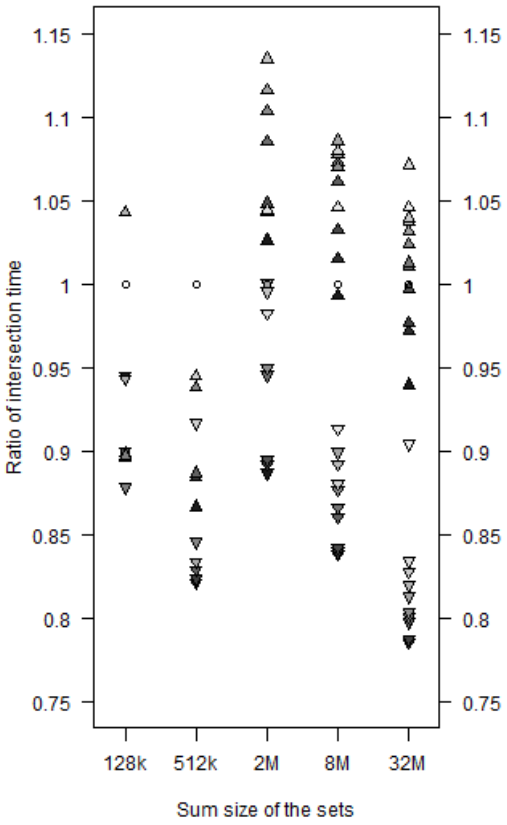



Chart 5.4: Binary search with look-up

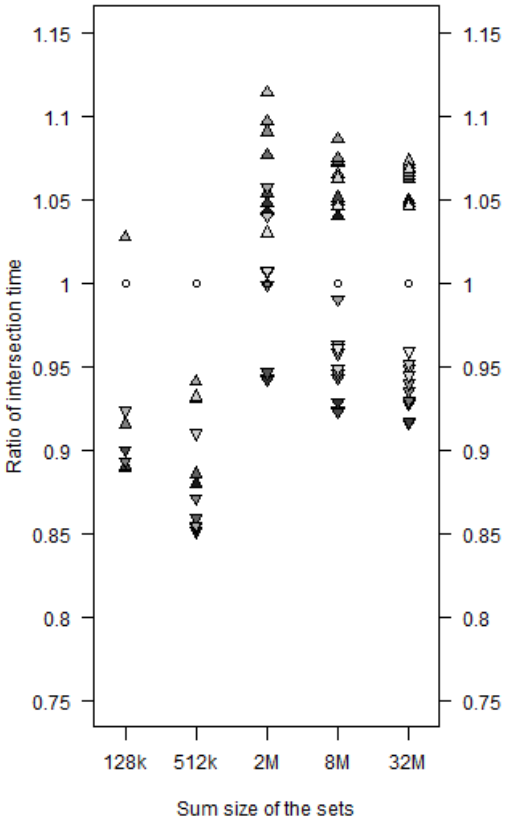



Chart 5.6: Interpolation search

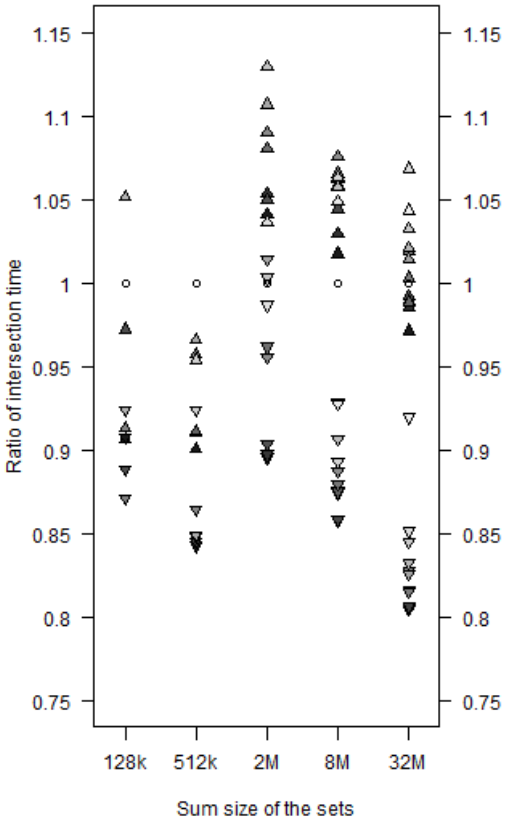



Chart 5.5: Interpolation search with look-up

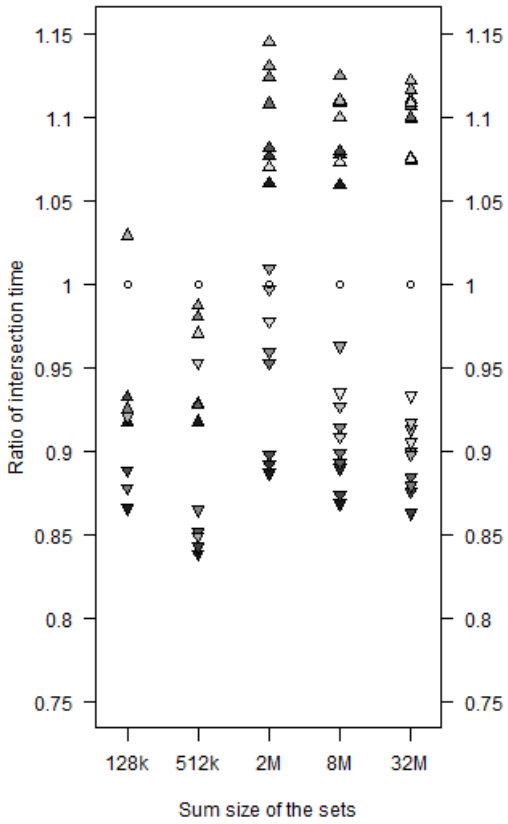



Chart 5.7: Generalized quadratic search

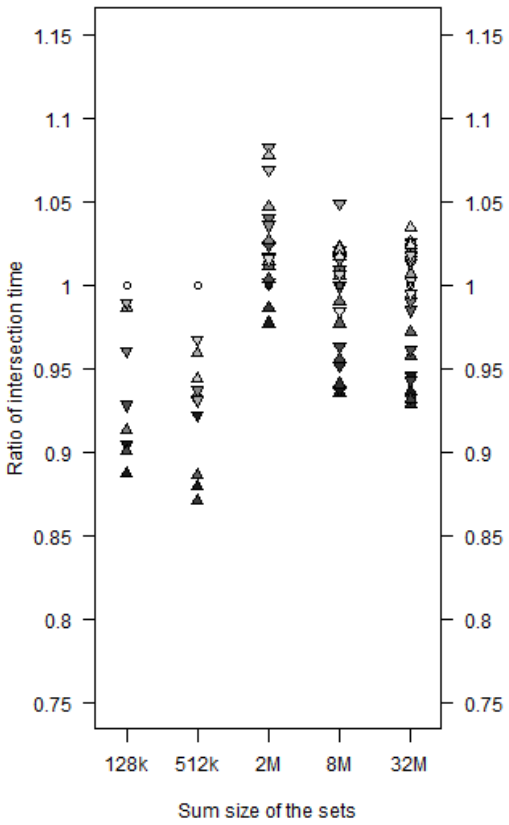
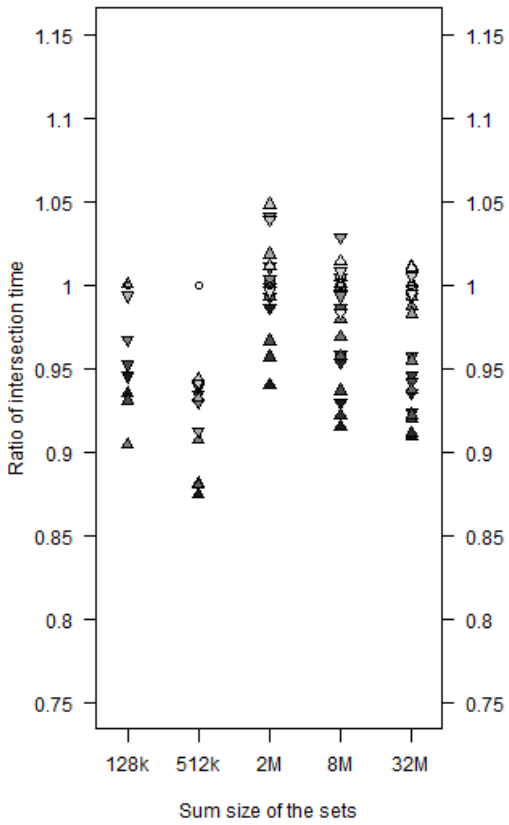



Chart 5.8: Local search (jobs created on CPU)




Graph 5.9: Local search (jobs created on GPU)

The local search strategies show symmetric results with regard to which set is smaller. This is expected as the algorithm is also symmetric, unlike the global search strategies. Here the maximum should be for ratio 1:1 – this does not exactly fit to our results but it can be caused by the properties of pair of sets used for our benchmarks.

## Comparison With CPU

The CPU merge-join is on a par with our GPU implementation. The CPU algorithm has lower theoretical complexity: the merge-join runs in linear time while our search algorithm has average time-complexity $O(n/p \cdot \log\log n)$. Moreover, the GPU algorithm has to copy data from RAM to GPU and back. Still, for sets with fewer common elements the GPU strategy can be more than 2× faster. There are two reasons for this: there are fewer elements to copy back from GPU, and the merge-join has to do approximately *2n* comparisons. However, as the sets get more common elements, the amount of data copied back to RAM grows and the number of comparisons in merge-join decreases to *n*. This results in worse performance of the GPU strategy.

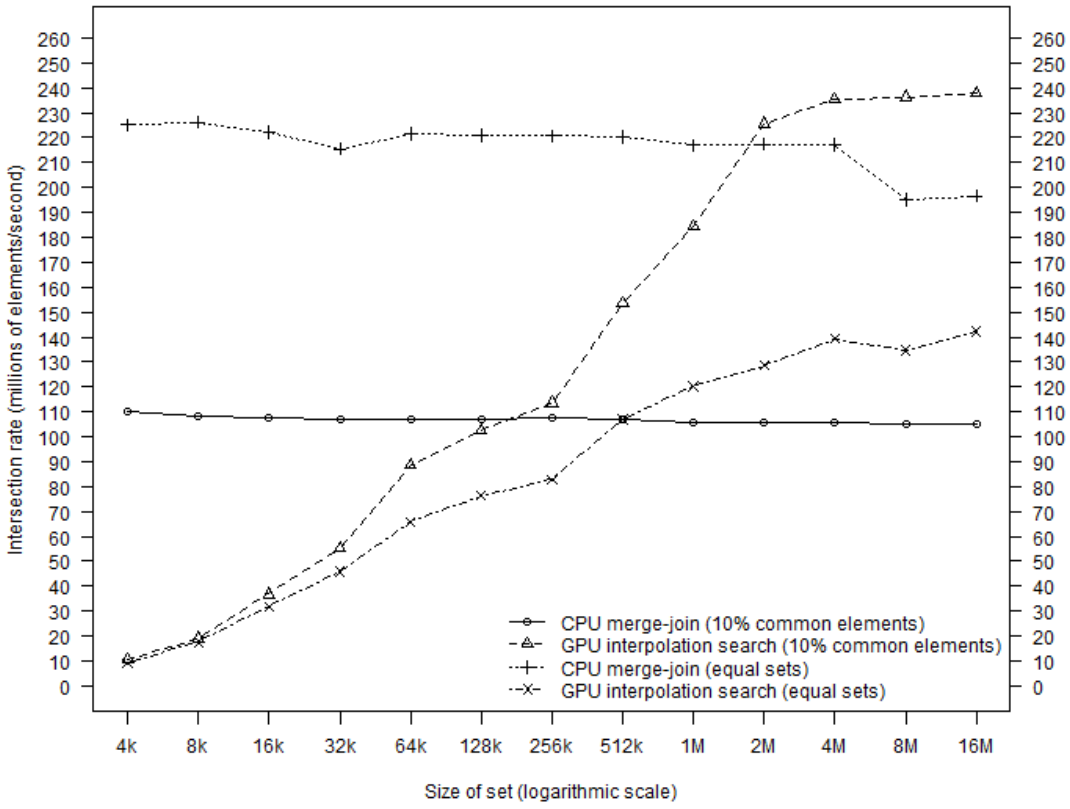



Chart 5.10: CPU vs. GPU on equally sized sorted sets

## 5.3 Hash-based Intersection

This section describes the algorithms which do not require the input sets to be sorted but one of the sets must be transformed into hash table.

### 5.3.1 Linear Hashing

The linear hashing (LHB) algorithm comprises of three phases. In the first phase, the hash-table is initialized with zeroes, in second phase each work-item loads one key from the first set and inserts it into the hash-table and in the third phase each work-item loads one key from the second set and queries the hash-table for its presence. If the key is found in the hash-table it is inserted into a local memory buffer. When all work-items in a work-group finish the query, this buffer is flushed into the output set in the global memory.

In the insert phase, a position is computed from the key using hash function $h(x) = x \bmod H$ where $H$ is the size of the table. When there is zero on the position in the hash-table, the element is simply written there. If a collision occurs, the position is increased by a number incommensurable with the size of the hash-table (113 in our case). This process is repeated until an empty position is found. As many work-items may try to insert an element on some position in a single moment, atomic functions must be used for accessing the hash-table.

When the hash-table is queried the key is hashed using the function $h(x) = x \bmod H$ and this position is probed. If there is zero, the algorithm ends. If it contains the key we are searching for, the key is added to the output set. If there is a key different from the queried key, the position is increased by 113 and the probe process is repeated.

As zero in the hash-table has the meaning of an empty position, the element 0 must be handled in a special way.

One way to optimize this algorithm is to keep the chains of keys in hash-table sorted – we denote this variation LHS. This can be done initializing the table with maximal numbers ($2^{32} - 1$) instead of zero. The atomic compare-and-swap operation is then replaced by atomic minimum operation[20] – the lower number is written into

20 *atom_min* function is a part of *extended atomics* OpenCL 1.0 extension while *atom_cmpxchg* is found in *base atomics* extension. Although NVidia GeForce GTX580 supports both extensions some devices may not. OpenCL 1.1 specification contains all these atomic function in the

the hash-table. If the original element is not $2^{32} - 1$ the insert operation continues on the next position (current position + 113) with the greater one from the inserted and original elements.

With zero replaced by $2^{32} - 1$ this number needs special handling instead of 0.

The query is also similar to the one used in the basic version of this algorithm. The difference is that the probe process does not end when zero is encountered but with any number greater than the queried key.

Initial loading keys from both sets is done with coalesced memory access. Copying found elements set from the local memory, where these are buffered, into the output set is also performed in the coalesced way with a minimal number of memory transactions. We have to synchronize work-groups by atomic incrementation of the output set counter – here may happen some collisions between work-groups. Nevertheless, the major bottleneck is the random global memory access using the atomic functions.

Our implementation uses the hash-table with size $H = 4 \cdot n$ as we are focused on performance rather than on the smallest memory footprint. With this load factor $\alpha = 0.25$, the number of accesses for the basic variant (without sorted keys) should be $(1-\alpha)^{-1} = 1.33$ when the key is not in the table and $\alpha^{-1} \cdot \log(1-\alpha)^{-1} = 1.15$ when the key is present [27]. In the sorted keys variation the number of accesses is equal for both successful and unsuccessful search and identical to successful search in the basic variant. Experimental results show that it is 1.16 – 1.38 for basic variant (the exact value depends on the number of common elements in the two sets) and 1.16 with sorted keys (without any dependency on the number of common elements).

### 5.3.2 Cuckoo Hashing

Cuckoo hashing was described for the first time by Pagh and Rodler [28] and recommended as GPU hashing algorithm by Alcantara et al. [21].

The hash-table consists of several columns (we have used 3), each equipped with its own hash function. When the key is inserted into the hash-table we choose one column and compute position of key in this column. The key *x* is inserted into this position regardless whether this position already contains another key or not. If

mandatory set of supported functions for 32bit memory access.

there was an empty space, the insert operation is finished. If there was another key *y*, it is replaced by *x* an the *y* is hashed into the next column. There it may replace another key *z*; the process is repeated in this case. That is why this is called cuckoo hashing: the new element always pushes the original element out of its nest.

With an unfortunate choice of hash functions, there is a chance that we find ourselves looping forever. This scenario is evaded by canceling the process of inserting into the hash-table after some maximal number of iterations. Then a new set of hash functions must be chosen and the table is completely rehashed.

The insert operation is shown in code listing 5.4.

```
for (i := 0; ; ++i) {
    column := i mod TABLE_WIDTH
    position := hash(key, column)
    key := swap[21](hashtable[column, position], key)
    if (key == 0) {
        break
    } else if (i >= MAX_LOOPS) {
        signalize failure
        break
    }
}
```

Code listing 5.4: Insert into cuckoo hashtable

The advantage of cuckoo hashing over linear hashing is in the query operation. The number of look-up operations is limited to the number of columns in the hash-table. No long chains known from linear hashing with higher table load factor may occur here.

This hashing algorithm has been implemented in two variations: In first (cuckoo hashing in the local memory – CHL), suggested by Alcantara et al. [21], both sets were divided into buckets small enough to fit into the local memory. Then each work-group processed single pair of buckets: it creates a cuckoo hash-table in the local memory from the first bucket and then queried it with keys from the second bucket. The second implementation ran cuckoo hashing directly in the global memory (CHG) in a way very similar to the linear hashing.

There is a question how to divide the set into buckets in the local memory implementation. We estimate the number of buckets necessary – in order to fit the

21 OpenCL basic atomics extension defines *atom_xchg* function for both local and global memory. Regrettably the version for local memory appears to not work with compiler/drivers version used, therefore it had to be replaced with a repeated *atom_cmpxchg* function.

bucket into the local memory the maximum number of keys in each bucket was set to $M = 1024$, therefore, we choose each bucket to contain $\lambda = 800$ keys in average. With uniform distribution of input keys and feasible hashing function[22] we can approximate the probability of one bucket overflow using Poisson distribution:

$$P(\textit{one bucket failure}) = 1 - \sum_{k=0}^{M} \frac{\lambda^k \cdot e^{-\lambda}}{k!} = 1.38 \times 10^{-14}$$

With the largest tested sets having $S = 2^{24} \sim 16 \times 10^6$ elements, separated into 20 972 buckets, the probability of overflow encountered when hashing any of the buckets is:

$$P(\textit{any bucket failure}) = 1 - e^{-P(\textit{one bucket failure}) \cdot \frac{2 \cdot S}{\lambda}} = 5.78 \times 10^{-10} \sim \frac{1}{1\,731\,119\,676}$$

The probability of failure depends on size of the whole set – we could programmatically adapt the average load factor to keep this probability constant, but such complication is not necessary for purposes of this thesis.

If any bucket overflows the maximum size *M*, we have to choose another hashing function and rehash both sets.

As the local-hashing implementation both loads and stores keys in a coalesced fashion, and the hashing itself is performed in the local memory, this part is fast enough. The bottleneck here is the bucketing, where we have to atomically increment a bucket size counter for each key itself and then write the key into the bucket, causing random accesses to the global memory.

### 5.3.3 Indexing into Large Bitmap

This method differs radically from the previous two algorithms. The hash-table size does not depend on the size of the set but on the size of the universum. As most present-day GPUs do not have enough memory[23] to keep the 512MB bitmap for the whole 32-bit universum, we have to split the sets into buckets according to first few bits. This strategy is called 'indexing into large bitmap with split' – ILBS. We have chosen 16 buckets for each set – one for each combination of the first 4 bits of the

22 Note that this hashing function is different from the one used for the cuckoo hashing in local memory.

23 The adaptation for GPUs with enough memory is straightforward.

key. This results in reducing the universum to 28-bit one for which only 32MB bitmap is needed.

In the first phase, we compute the required sizes for buckets and reserve memory for them. Then in phase II the sets are split into the buckets – work-groups keep a buffer in the local memory for each bucket and when some buffer is full, it is flushed into the bucket in the global memory. As the work-items cooperate on the flush there is only one increment of bucket counter for all keys in the buffer. The memory transfer is also coalesced.

Third phase contains a loop over all pairs of buckets. In each of 16 iterations this process is repeated:

1. The 32MB bitmap is filled with zeroes.
2. Keys are loaded from the first bucket of the *i*-th pair of buckets.
3. A bit is written using atomic OR into the bitmap on the position specified by lower 28bits of each key.
4. Keys are loaded from the second bucket of the *i*-th pair of buckets
5. The bitmap is probed for a bit on position specified by lower 28bits of each key.
6. If the bit was 1 the key is added to the output set.

We have also implemented other version of the algorithm, called 'indexing into large bitmap – no split' - ILBN. The keys are not split into buckets; when the bitmap is constructed the keys whose higher 4bits do not match the current mask are simply ignored. Therefore, each key is loaded 16× instead of 2× as in the variation described above, but there is no bucketting phase with additional required memory. We will see how this affects the perfomance in the results below.

### 5.3.4 Bloom Pre-filtering

Bloom filter [29] is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. As false positives are possible but false negatives are not, it allows to eliminate some elements from the sets which cannot be in the intersection.

The idea of Bloom filter is to hash element to a position of a bit in a bitmap. The hashing function is usually trivial such as ignoring some bits in the element. Collisions may occur:

- if there is 1 in the bitmap, the element **may** be hashed into the filter
- if there is 0 in the bitmap, the element **was not** hashed into the filter

The algorithm using is data structure has been used as a template – at first the sets are reduced and then other non-probabilistic algorithm is executed. The reduce algorithm comprises of a loop with these steps:

1. Create the Bloom filter from first set.
2. Filter the second set through the Bloom filter into third set.
3. If the third set is larger than 70% of the first set, quit.
4. Rename the first set to second set, third set to first set and continue with next cycle of the loop.

The Bloom filter may be either located in the global memory, (almost) not limiting its size but with slow random access, or in the fast local memory, although very limited to its size. In our implementation the global filter uses *N/4* bytes of memory with *N* elements in the set; the local filter has fixed size 14336 bytes, which is the maximum amount of the local memory assignable to one work-group with optimal occupancy.

The global filter is created using atomic OR operations in the global memory, construction of local filter is a bit more complicated:

1. Global filter filled with 0s is constructed in the global memory, sized equally to local filters.
2. Each work-group creates its own filter in the local memory, hashing its part of the set into the local filter.
3. Local filters are ORed (word-by-word) to the global filter.
4. Each work-group fetches the finished global filter into the local memory and then reads only from this local copy.

### 5.3.5 Results

In this section we will compare the GPU strategies with each other, then analyze the effect of Bloom prefilter on each of them, and present the results for non-equally sized sets. In the end we will see the final comparison with CPU intersection.

#### GPU Strategies Comparison

Similarly to the benchmark of sorted sets intersection, we have used sets sized from $2^{12}$ = 4k to $2^{24}$ = 16M keys and configurations with no common elements, 0.1%, 10%, 50%, 90% and identical sets (differently shuffled, of course). We also omit most of the graphs and present only those with some interesting characteristics. For the rest of the charts please refer to the enclosed CD.

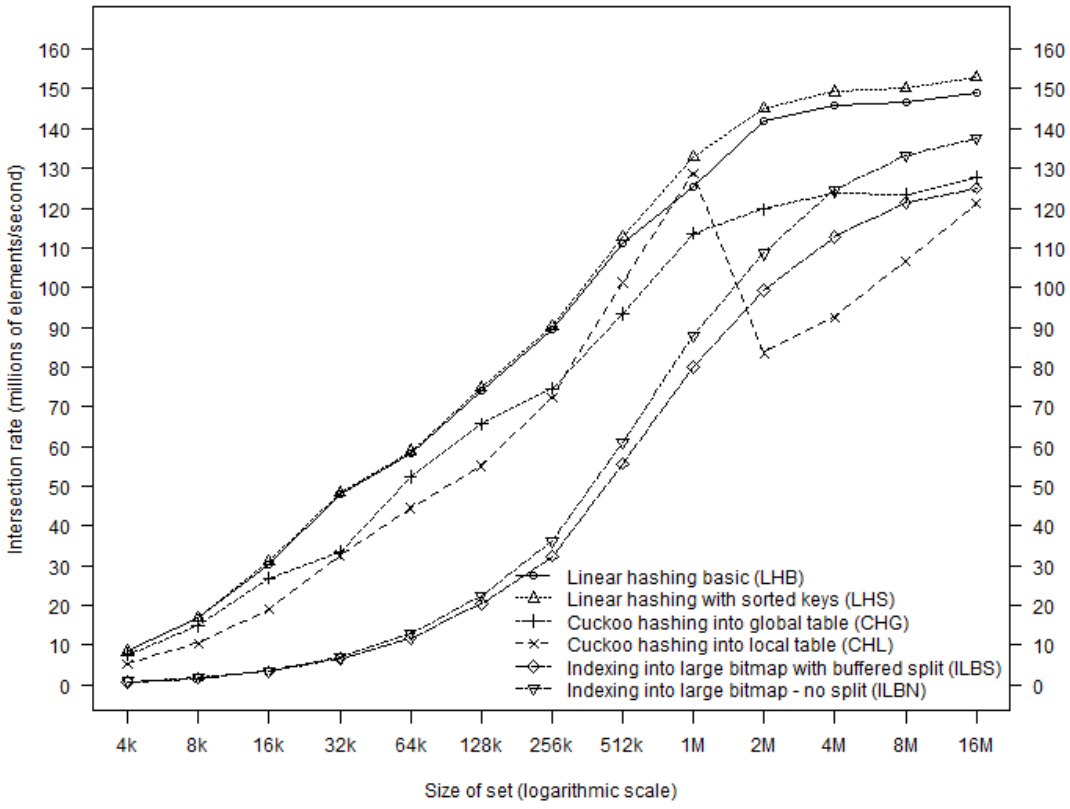



Chart 5.11: GPU strategies on equally sized unsorted sets with 10% common elements

One of the simplest algorithms – the linear hashing with sorted keys – is the fastest one for all sizes of sets. The performance of basic linear hashing is very similar to the sorted-keys version, especially with more common elements.

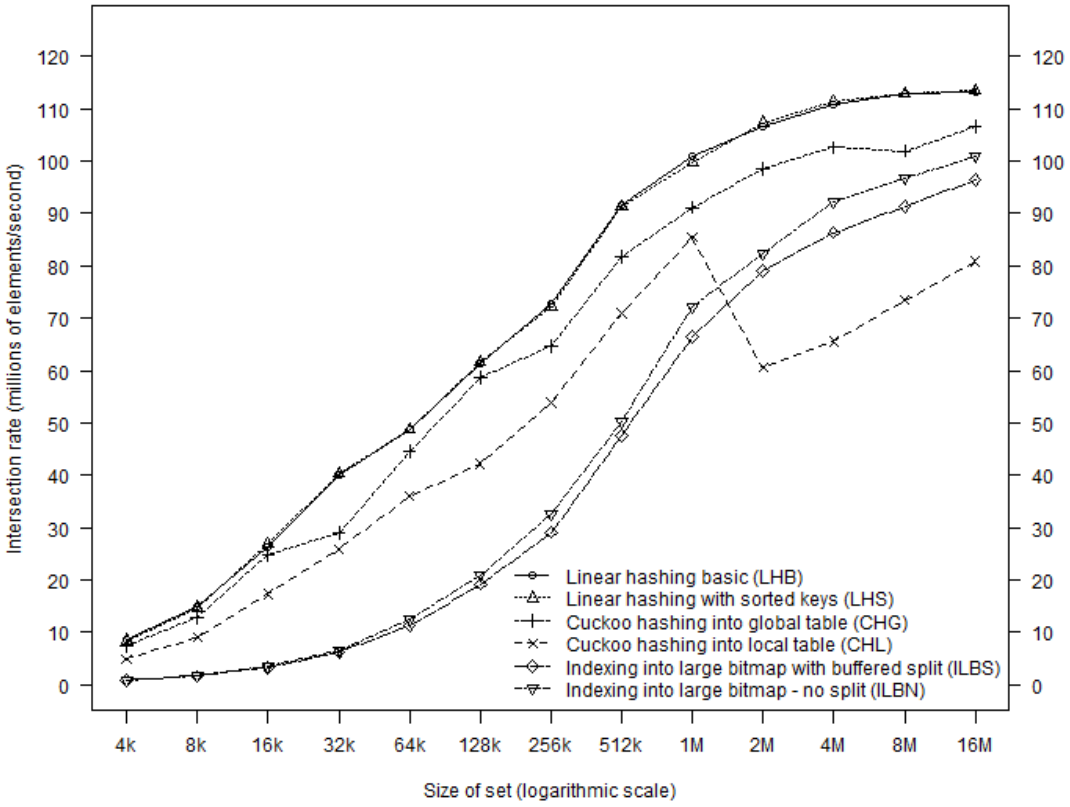



Chart 5.12: GPU strategies on equally sized unsorted sets with 90% common elements

Although the cuckoo hashing into a global hash-table has very good results for many common elements, when there are only few common elements it requires more memory accesses than the linear hashing, and therefore, it has only mediocre performance.

The performance of cuckoo hashing to local table significantly deteriorates for sets larger than 1M elements. There is no obvious outer reason for this – probably some hardware resources stop scaling at this moment. This deterioration is found in all results across the configurations with little deviation. Exact origin of this behavior was not located, although L2 cache could be relevant because of its size 768kB.

Filling a large area of memory with 0s, whose size is not dependent on the size of input, renders unsurprisingly both strategies with indexing into this bitmap rather slow on small input sets. With larger sets the constant work requires smaller relative part of the computation, and finally, the overall performance is moderate but still far from the optimal case.

## Bloom Pre-filters

The result of applying Bloom prefilters to other strategies can be seen in charts 5.13 and 5.14. With none or only few common elements, the prefiltering was able to reduce the sets so that running the inner strategy was almost unnecessary – the few lasting elements could be processed on CPU. In some cases the reduction itself outperformed the actual strategy. Nonetheless, this did not happen for all strategies – in our results linear hashing is always faster without the Bloom pre-filter.

The local memory is sufficient for sets with at most 128k elements. Larger sets make the local bitmap too dense with 1s and the fast accesses cannot compensate this anymore. Larger bitmap in the global memory is much more suitable for these sets.

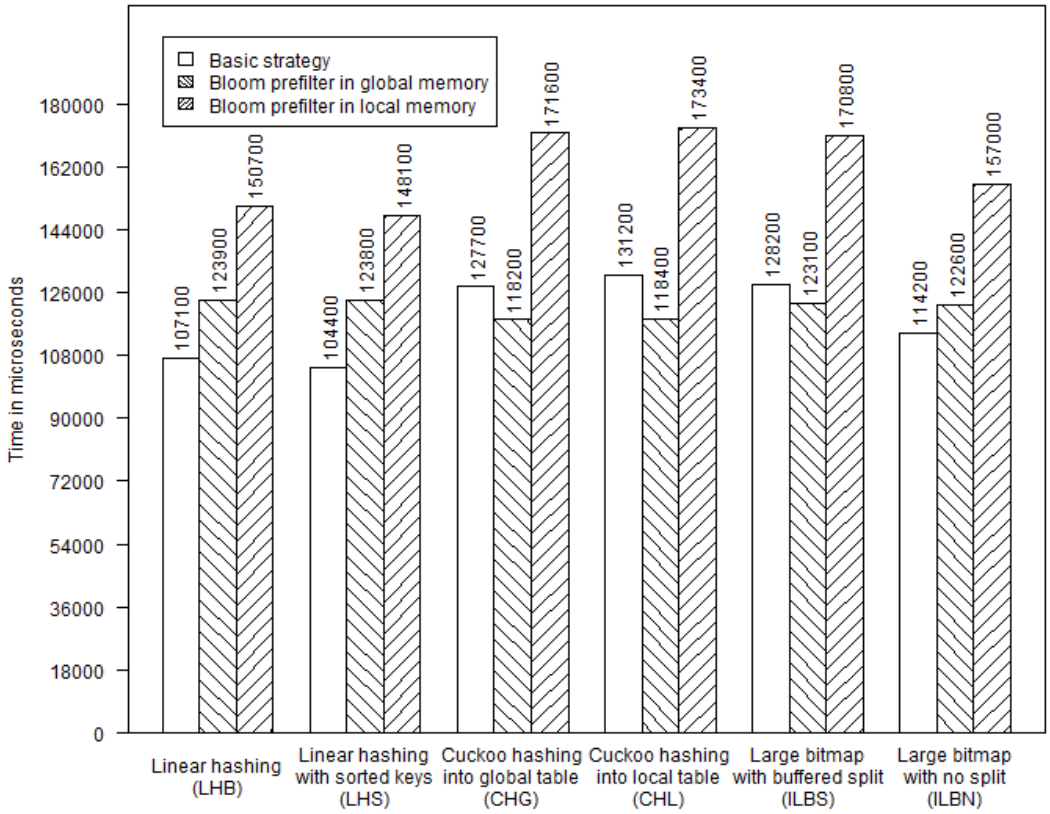



Chart 5.13: Bloom pre-filters for 16M unsorted sets and 0.1% common elements

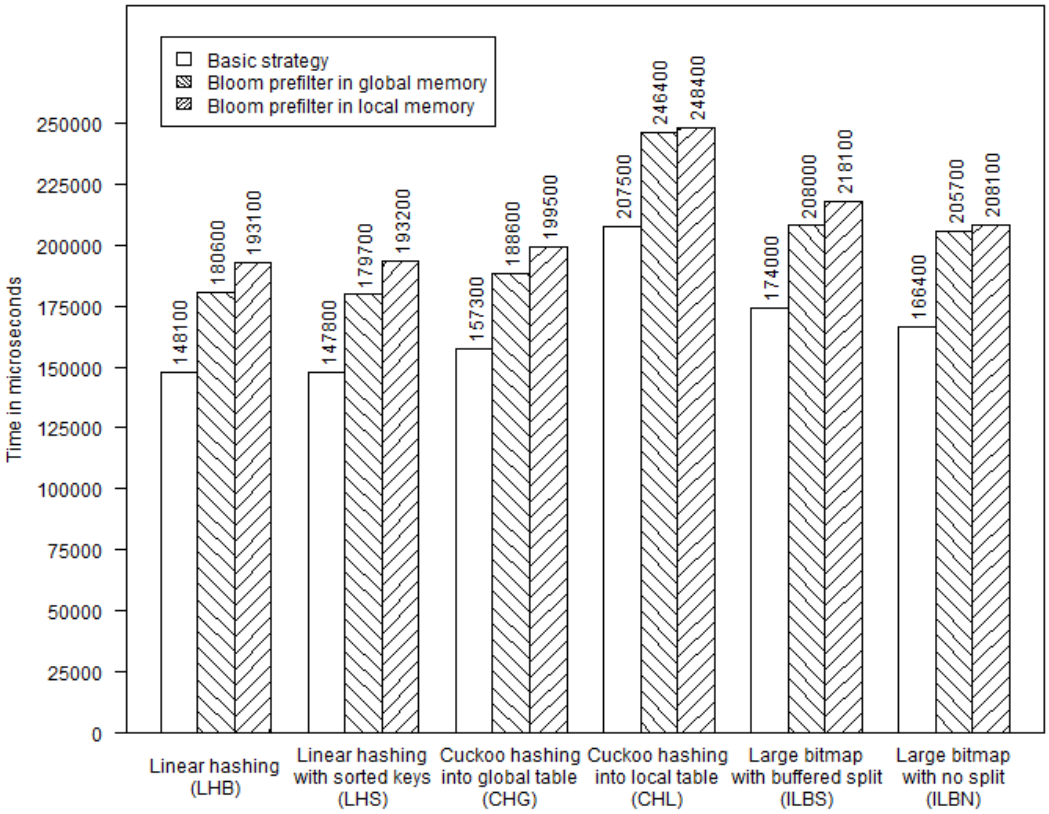



Chart 5.14: Bloom pre-filters for 16M unsorted sets and 90% common elements

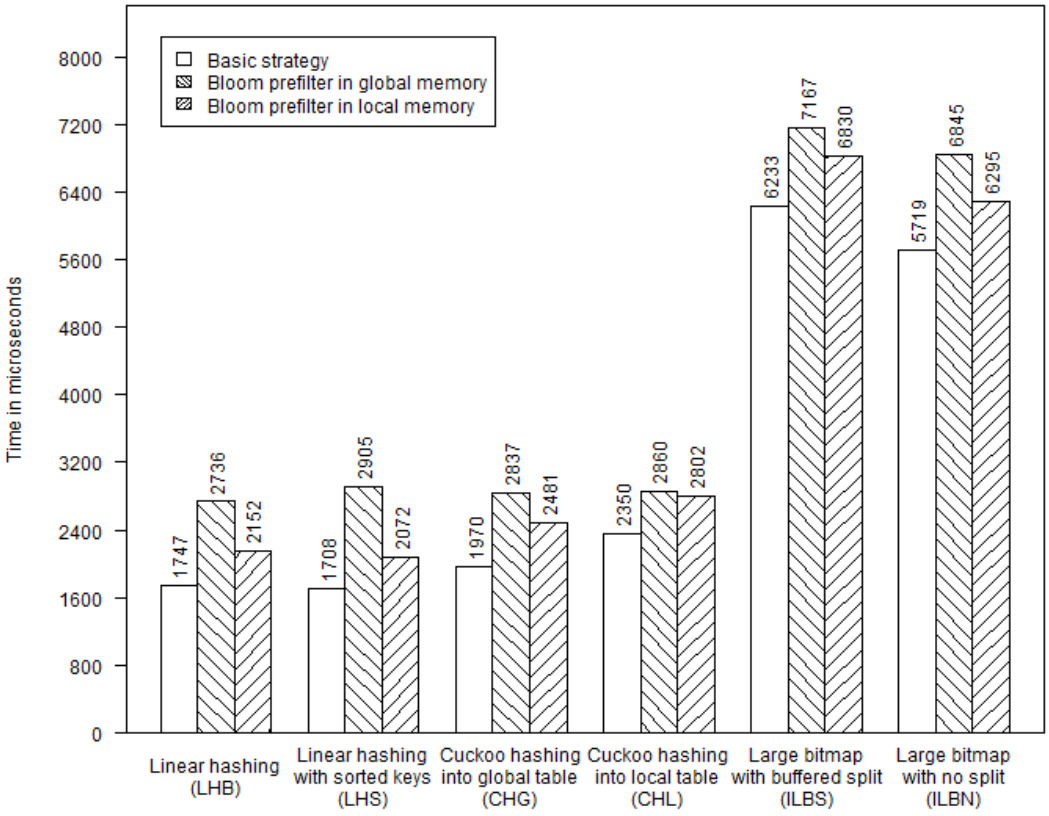



Chart 5.15: Bloom pre-filters for 128k unsorted sets and 0.1% common elements

## Asymetric Sets

We have used the same methodology for identifying the behavior of hash-based intersection algorithms on asymmetric sets as previously with algorithms running on sorted sets. The sets in pairs had ratios in the form $1:(2^i - 1)$ and there were 4k elements common to both sets.

Again, execution times in the charts below are normalized to the execution time of pair with equally sized sets, marked as empty circle. If the triangle points upwards the first set passed to the algorithm is greater, if it points down the second set is greater. The darker the triangle is the greater is the difference between the sizes of the two sets.

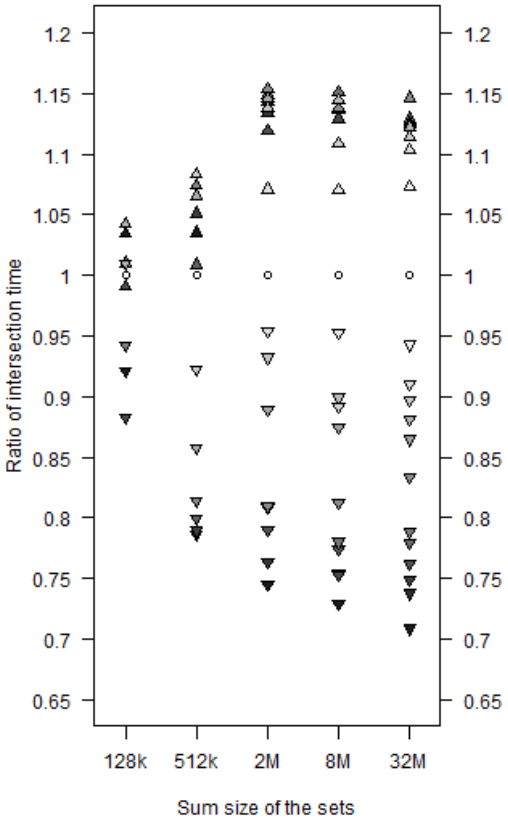



Chart 5.16: Basic linear hashing (LHB)

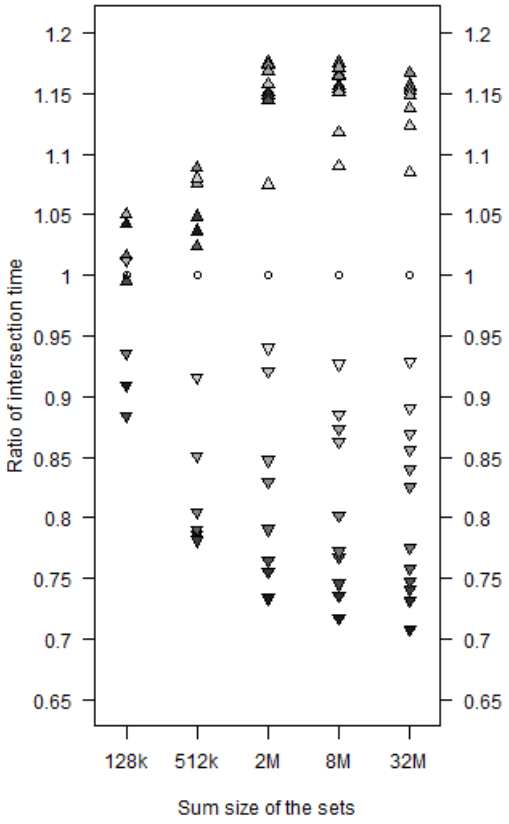



Chart 5.17: Linear hashing with sorted keys (LHS)

As the pairs with larger first set have longer execution times we can deduce that in linear hashing strategies (LHB and LHS), building the hash-table is the most expensive operation. The look-up into this hash-table is much cheaper.

Cuckoo hashing to the global memory (CHG) show similar results as the linear hashing – the pairs with a large second set are executed faster than those with larger

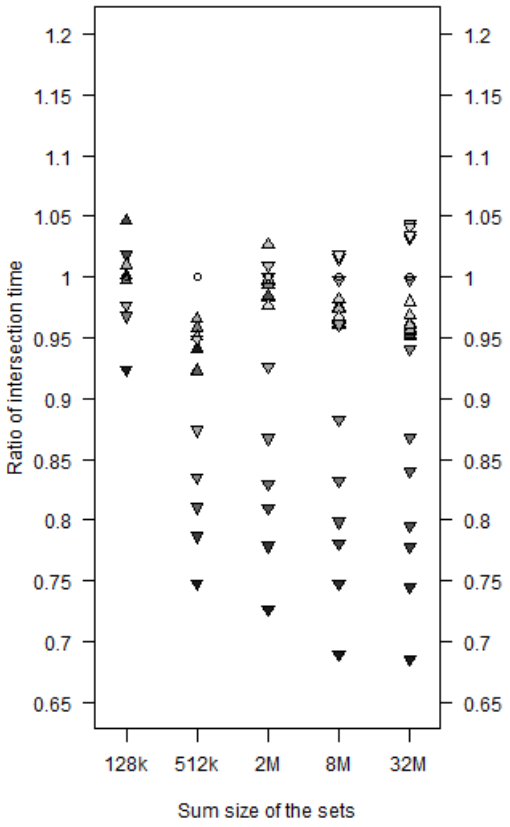



Chart 5.18: Cuckoo hashing to global memory (CHG)

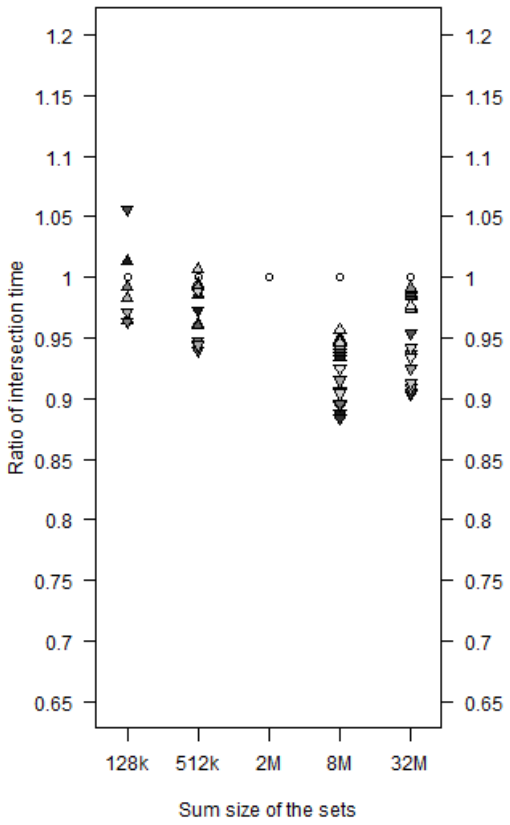



Chart 5.19: Cuckoo hashing to local memory (CHL)

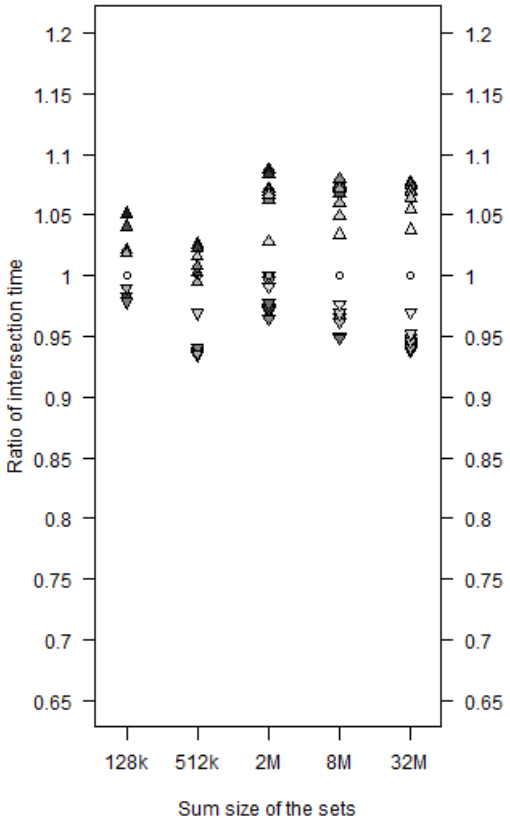



Chart 5.20: Indexing into large bitmap with buffered split (ILBS)

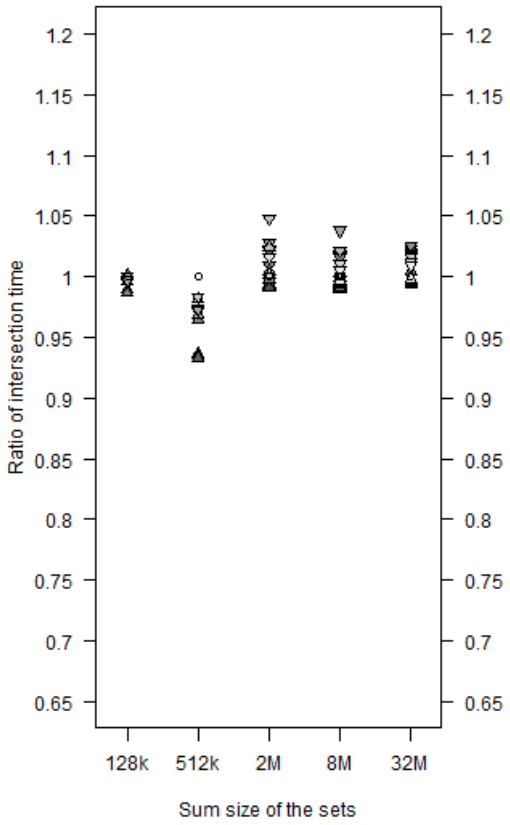



Chart 5.21: Indexing into large bitmap - no split (ILBN)

first set. However, the difference between larger first set and both set equal is not as prominent. The exact nature of the first set has higher impact on the overall execution time.

On the other hand, cuckoo hashing to the local memory (CHL) shows little difference between symmetric and asymmetric sets. Here the bucketing takes the majority and the same actions are performed on elements from both sets. Still, we can observe somewhat shorter execution times with larger second set and vice versa. We should note that the missing comparison on 2M total size is caused by extremely low execution time of the equally sized sets variant – all the other sets have ratio around 1.8.

Building the bitmap in IBBS strategy is a bit more expensive than the look-up as we use atomic instructions for this purpose, but the difference is not as significant as in linear hashing – here it is at most 10% on each side. In the version without split (IBBN) the difference is yet smaller, about 2%.

### Comparison with CPU

There are multiple algorithms for intersection on CPU as well. We use serial merge-join with sets sorted by tbb::sort[24] (having good performance on smaller sets) and two-pass bucketing developed by Kruliš [30] (excelling for greater sets). As the decision can be based solely on the input size, the appropriate CPU strategy for the particular set can be always selected in advance. This is why the number of buckets used in our results of two-pass bucketing is varying between 32 and 1024 – we have collected results for all settings and selected the optimal values.

GPU outperforms both CPU strategies for sets larger than 16k. The tbb::sort with sequential merge-join is better for pairs of sets with up to 2M – 4M keys each, after that the two-pass bucketing algorithm, finally, starts scaling. However, even at its peak[25] performance for ~128M sets with rate around 70M keys/seconds this CPU algorithm cannot compete with GPU linear hashing.

24 Already referenced in section 4.3.4.

25 These data are not a part of the graph as the GPU linear hashing strategy is not suitable for such large data sets.

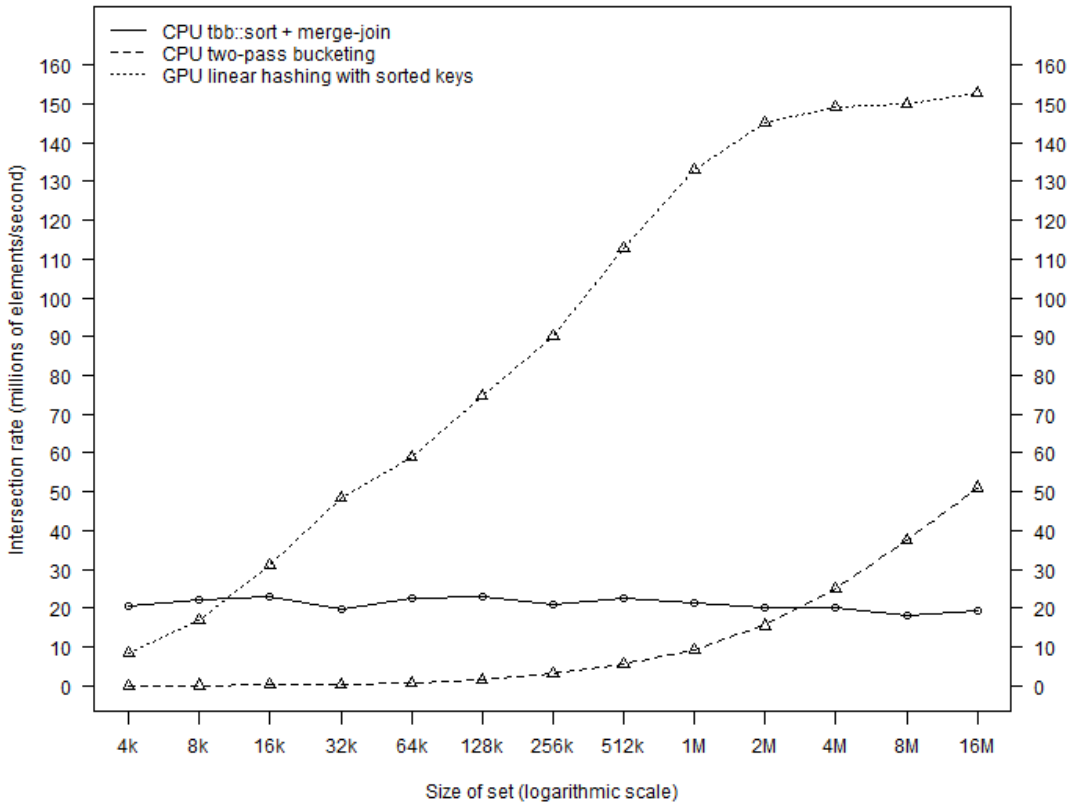



Chart 5.22: CPU vs. GPU on unsorted sets with 10% common elements

# 5.4 Sets Not Fitting into Memory of GPU

The GPU memory is usually smaller than the RAM memory. Moreover, there are also limitations for the size of a single continuous block of memory. That it is why we have to develop different techniques for larger sets that cannot be present in the GPU memory at one moment.

Using secondary storage is not a new technique. It has been used for years in many external memory algorithms, for instance in mergesort as the best known example. In the common CPU case the RAM memory is used as the primary storage and magnetic hard drives or solid state drives second, we use the GPU memory as the primary and RAM as the secondary storage.

The techniques below usually try to overlap GPU computation with memory transfer because these operations are independent.

## 5.4.1 Splitting into Multiple Partitions

The naive approach to calculating the intersection of a pair of too large sets is to partition both sets to subsets of suitable size and intersect each subset from first set

with each subset from second set (this is basically the NINLJ idea). The output set is plain concatenation of the partial results. However, as this has quadratic time-complexity with respect to the number of the partitions, we mention this strategy rather for a completeness – it has no practical purpose for us.

If the sets are not sorted in any way we have to stick to bucketing. Similarly to the assumption described in section 5.3.2, we set the expected fill to 80% of the max size we can process on GPU. This means that we use $N/(M \cdot 0.8)$ buckets. Yet worse than in section 5.4.2, both sets must be processed before the execution on GPU begins. Then some inner hashing strategy[26] is executed on each pair of buckets. We call this outer strategy 'multirun bucketing template' (MBT) because it is not dependent on the inner strategy.

Inserting values into the buckets one-by-one would cause effect known as cache-line ping-pong; prior to inserting element into the bucket the cache-line that should contain this element would have to be transferred to the processor which tries to write into the bucket. Therefore, we use a common technique to avoid this behavior – we store the elements in thread-local buffers and do the insertion only after any buffer gets full.

When the sets are sorted the situation is better. We can partition the sets without any bucketing – we do not use hashing but find sequences with the same range. The routine is executed serially, as we need usually only several partitions, trying to do this in parallel would be excessive – the overhead related to parallelization would be too high. The splitting algorithm is outlined in code listing 5.5. This strategy is denoted by MST ('multirun sorted template').

After the partitioning is finished, each pair of subsets is intersected by some sorted set intersection strategy described above.

In order to parallelize copying of the data to GPU and back and execution of kernels, multiple command queues are used, each managed by a separate thread. However, according to results in profiler, the OpenCL implementation was not able to exploit this command pattern.

26 We use the linear hashing with sorted keys (LHS) as this has proven as the fastest strategy in previous results.

```
beginA := begin of setA;
beginB := begin of setB;
while (beginA != end of setA && beginB != end of setB) {
    glbA := beginA + maxDataSize;
    if (glbA < end of setA) {
        glbB := GLB of setA[glbA]
                                        from beginB to end of setB
    } else {
        glbA := end of setA
        glbB := end of setB
    }
    if (glbB - beginB > maxDataSize) {
        glbB := beginB + maxDataSize;
        glbA := GLB of setB[glbB]
                                        from beginA to end of setA
    }
    intersect setA[beginA .. lowerBoundA]
                                    with setB[beginB .. glbB]
    beginA = glbA;
    beginB = glbB;
}
```

Code listing 5.5: Partitioning of sorted sets

### 5.4.2 Indexing into Large Bitmap

The idea of this algorithm is thoroughly described in section 5.3.3, but the implementation on CPU slightly differs. The sets are not transferred to GPU as large blocks which would not fit to the GPU memory but these are partitioned into several smaller subsets. The strategies below differ in the way how these smaller subsets are created.

The commands to copy the subsets from host (RAM) to GPU, and execute kernels hashing them into the bitmap, are sent to out-of-order command queue with event-based synchronization – we have to wait before the subset may be hashed until it it is present in the GPU memory. The same applies for subsets from the second set and kernels querying the bitmap and building output set.

In this scheme, the GPU can theoretically both copy data and execute kernels. However, according to the profiler, this opportunity was wasted.

We have implemented this algorithm in two variations, each partitions the data in a different way. The simpler version 'multirun large bitmap simple strategy' (MLBS) similarly to IBBN does not perform any preprocessing on CPU. We always insert only elements from one of the 16 ranges into the bitmap, therefore, the algorithm comprises of 16 loops. In each loop we send blocks of memory from

the first set (represented as continuous memory area) to the GPU. All elements which fit into the currently processed range are inserted into the bitmap. Then the second subset is also sent to the GPU in the same way, performing a query into the bitmap only for those elements which fit into the processed range.

In this version all the data are sent to GPU 16×, which requires a considerable amount of time. The transfer is ineffective because each element is actually utilized only once but it is ignored 15×. However, there is almost no CPU computing power required.

The alternate variation 'multirun large bitmap bucketing strategy' (MLBB) uses multiple CPU threads to split each set into 16 buckets. Then each bucket is sent to the GPU only once. Using more computing power on CPU is a trade-off for reduced memory throughput. Note that this is exactly the same as we did in ILBS strategy, only the first phase is processed on the CPU instead of the GPU.

As we want to evade excessive synchronization each thread has 16 small thread-local buffers. The thread processes one part of the set and buckets the elements into its local buffers. When any of the buffers becomes full enough it is handed over to one of 16 shared lists[27] of ready buffers which collect the processed elements – these lists represent the buckets. There is another special thread which sends these buffers to the GPU and manages the execution of kernels that insert elements into the bitmap or query it.

We can always keep only single bitmap in the memory. That is why the elements from all lists cannot be sent in parallel. The management thread can start sending data from the first list of first set anytime but it has to send all data logically belonging the this list before continuing – this is not until the whole first set is processed. Then it can start sending data from first list of second set and execute kernels querying the bitmap. After both first lists are processed we can start with second list of first set and so on – now the bucketing threads are already finished.

In fact there are only two moments where manipulation with GPU is parallel to the CPU processing. The first is in partitioning the first set – it is not required to have the whole first bucket processed before starting to send the data from this bucket to

27 The list is implemented by TBB concurrent queue.

GPU. The second one is in bucketing the second set on CPU while the GPU inserts the elements from first set into the bitmap.

### 5.4.3 Results

We have tested the strategies on pairs of sets, each having up to 512M elements whereof 10% common to both sets. The maximum size of one subset generated from the large set was set to 4M as we are usually manipulating with multiple subsets in parallel, allowing some parallelism in the memory transfer and processing.

#### Intersection of Sorted Sets

We have developed only single strategy for intersection of sorted sets but as this strategy is applied on other strategies, we provide results with the specific implementations of the inner strategy.

We do not show results with the interpolation search strategy (ISS) as the inner strategy because it has proven not fully operational. Despite the strategy provided correct output, the execution times were absolutely beyond expectations – in the slower way. The reason was not found but we have committed several experiments, varying the parameters of this strategy.

One explanation would be that the initial approximation of left and right key to 0 and $2^{32} - 1$ is not fitting anymore with the different range. However, the error in initial range should be corrected after first two iterations. That would cause constant slowdown for each processed element, not dependent on the number of partitions. We have varied the size of subsets, changing the number of partitions. According to our observation the execution time is roughly proportional to the number of subsets created from one set. Therefore, the initial range error cannot explain such behavior.

We have also tried slower variants with correct initial range – the elements on both borders of the subset were actually loaded instead to the initial approximation of 0 and $2^{32} - 1$. Then the execution times were substantially different for each setting of the subset size but we have not observed any pattern in this behavior.

Any error in implementation is eliminated – the inner strategy code is used also in the benchmark of smaller sets, and the templating strategy is used for other search strategies without any modifications. After all we omit the results of this strategy as untrustworthy.

The look-up version of binary search strategy is also omitted as this had not improved the performance in the previous benchmarks.

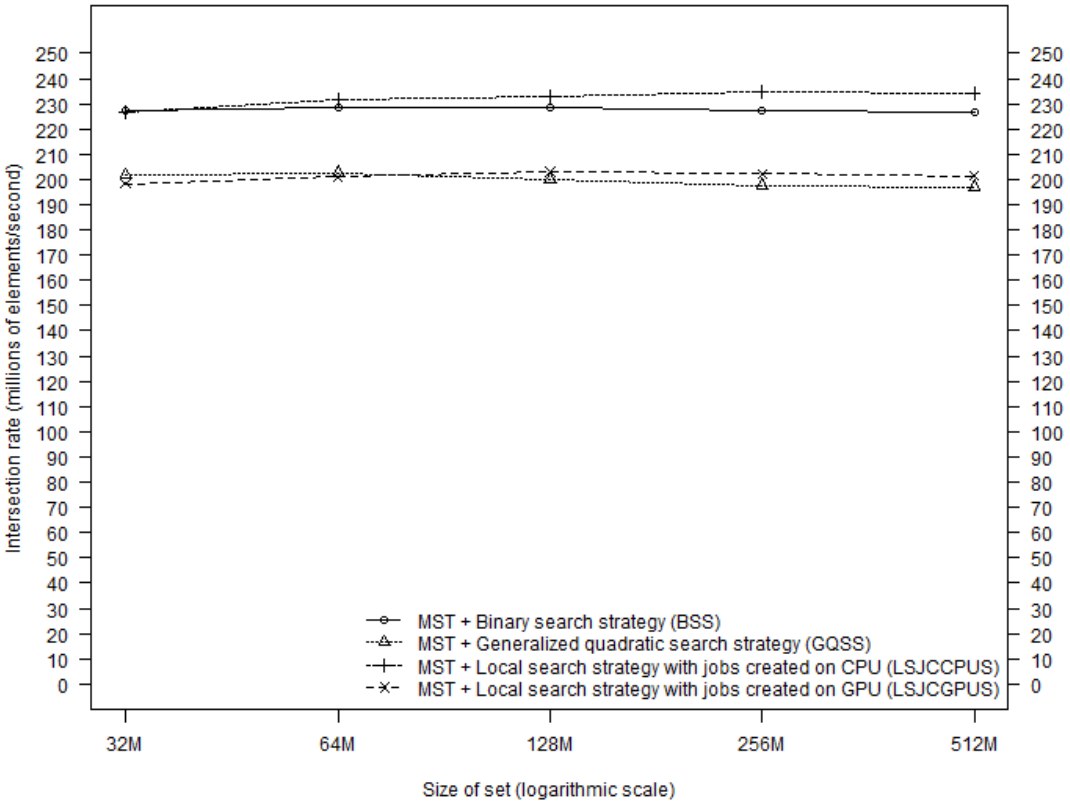



Chart 5.23: Multirun sorted template strategy

The results above correspond to our expectations. Our solution scales well regardless of the size of sets. For reasons mentioned above, the local search strategy with jobs created on CPU does not compete with interpolation search strategy and therefore is the fastest one, followed by binary search strategy. The other two strategies provide slightly worse results. This is similar to the results we have obtained in previous sections.

## Hash-based Intersection

Here we present the bucketing strategy applied on linear hashing with sorted keys (as the fastest strategy on small sets) and both types of indexing into large bitmap.

The only useful strategy is the indexing into large bitmap with sets bucketed on the CPU prior to sending the data to GPU (MLBB). The intersection rate is lower than those achieved with smaller sets (there we had rate about 120 – 155 million keys per second) but still significantly higher than the rate we measured for CPU-only algorithms.

Both indexing into large bitmap without the initial bucketing (MLBS) and applying multirun bucketing template on linear hashing with sorted keys (MBT + LHS) showed unsatisfying performance, comparable with tbb::parallel_sort followed by merge-join.

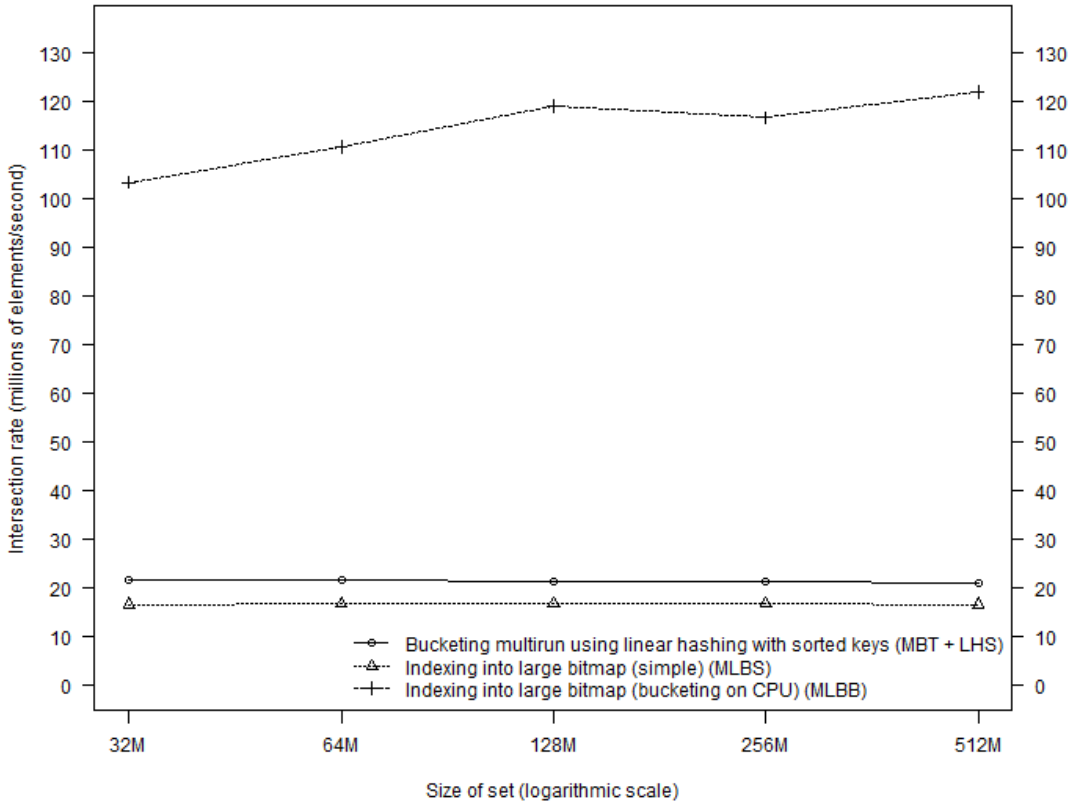



Chart 5.24: Hash-based strategies on unsorted sets

# 5.5 Future Work

We have studied the simple form of intersected data and assumed that we process each set only once. Therefore, we could not use more time and memory to pre-process it to any form that would help us perform the intersection faster. If there would be more requests to intersect one static set with multiple another ones, it could be useful to sort it or build a search tree to accelerate the further queries. In this case, we could also amortize the cost of building a hash-table.

In real-world databases the indices are sometimes compressed with keys represented using delta-encoding[28]. Such data representation might require non-

28 The keys are stored not by their value but by difference to previous key. This representation usually requires less bits but requires special handling when the difference is out of the expected range.

trivial modification of the algorithms, having a significant effect on the performance, but allowing larger sets to be stored directly in the GPU memory.

Another variation of the problem is the intersection of multiple sets. Aside from not transferring the output sets back to RAM and forth to GPU, efficiency of various set pair selection strategies, or techniques comparing multiple sets at one moment, could be analyzed.

We could also test scalability of our algorithms on systems with multiple GPUs. The versions suited for large sets could be easily modified for such system configurations.

# 6. Conclusion

In the presented work, we have provided implementations of several algorithms for sorting and set intersection. The main objective was to decide whether these algorithms can be efficiently implemented on GPGPUs and which of them are the most suitable for this purpose. This work brings an extensive comparison of those algorithms with worthy results.

Although not all algorithms were efficiently portable to GPU and some of our attempts for optimization did not succeed, we have always found a way how to solve the problems of sorting and set intersection faster on GPU than on CPU. We have also presented solution how to overcome present-day hardware limitations of GPUs, although the OpenCL library did not enable us to fully exhibit some techniques.

We have not used data from real database systems and implementing a business ready system would require a lot of effort. Nevertheless, we provide our implementation of various algorithms on the enclosed DVD, and through our comprehensive results offer a valuable advice for any developer considering usage of GPU in data-processing systems.

# Bibliography


[1] KEANE, Andy, *"GPUs Are Only Up To 14 Times Faster than CPUs" says Intel* [webpage], http://blogs.nvidia.com/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel/, 2010

[2] HRISTOV, Vassil, *Performance Evaluation of Query Processing Algorithms on GPGPUs*, School of Informatics, University of Edinburgh, 2010

[3] BAKKUM, Peter - SKADRON, Kevin, *Accelerating SQL database operations on a GPU with CUDA*, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units , Pittsburgh, PA, USA, 2010

[4] Khronos OpenCL Working Group, *The OpenCL specification 1.1 rev. 44*, http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2011

[5] KHRONOS group, *KHRONOS OpenCL website* [webpage], http://www.khronos.org/opencl/, 2012

[6] NVidia, *NVidia CUDA C Programming Guide, version 4.1*, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2011

[7] NVidia, *OpenCL Programming Guide, version 4.1*, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf, 2012

[8] NVidia, *OpenCL Best Practices Guide*, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Best_Practices_Guide.pdf, 2011

[9] LARABEL, Michael, *Looking At The OpenCL Performance Of ATI & NVIDIA On Linux* [webpage], http://www.phoronix.com/scan.php?page=article&item=opencl_nvidia_ati, 2010

[10] GORESKY, Mark - KLAPPER, Andrew, *Fibonacci and Galois Representations of Feedback with Carry Shift Registers*, IEEE Transactions on Information Theory, 2002

[11] GOVINDARAJU, Naga K. - RAGHUVANSHI, Nikunj - HENSON, Michael - TUFT, David - MANOCHA, Dinesh, *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors*, http://gamma.cs.unc.edu/GPUSORT, 2005

[12] GREß, Alexander - ZACHMANN, Gabriel, *GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures*, 20th IEEE International Parallel and Distributed Processing Sym-posium (IPDPS), Rhodes Island, Greece, 2006

[13] CEDERMAN, Daniel - TSIGAS, Philippas, GPU-Quicksort: A practical Quicksort algorithm for graphics processors, 2009

[14] SATISH, Nadathur - HARRIS, Mark - GARLAND, Michael, *Designing*

*efficient sorting algorithms for manycore GPUs*, Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 2009

[15] MERILL, Duane - GRIMSHAW, Andrew, *High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing*, Parallel Processing Letters, 2011

[16] HOARE, Charles Antony Richard, *Quicksort*, The Computer Journal, 1962

[17] BATCHER, Kenneth E., *Sorting Networks and their Applications*, Spring Joint Computer Conference, AFIPS Proc, 1968

[18] KNUTH, D.E., *Sorting by Merging*, *The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed.* , p.158-168, 1998

[19] Intel Corporation, *Threading building blocks* [webpage], http://threadingbuildingblocks.org/, 2012

[20] RESEN, Rasmus - PAGH, Rasmus, *A New Data Layout For Set Intersection on GPUs*, IPDPS '11 Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, Anchorage, Alaska, 2011

[21] ALCANTARA, Dan A. - SHARF, Andrei - ABBASINEJAD, Fatemeh - SENGUPTA, Shubhabrata - MITZENMACHER, Michael - OWENS, John D. - AMENTA, Nina, *Real-Time Parallel Hashing on the GPU*, SIGGRAPH Asia '09 ACM SIGGRAPH Asia 2009, Seoul, South Korea, 2009

[22] BINGSHENG, He - YANG, Ke - FANG, Rui - MIAN, Lu - GOVINDARAJU, Naga - QIONG, Luo - SANDER, Pedro, *Relational Joins on Graphics Processors*, 2008 ACM SIGMOD international conference on Management of data , Vancouver, BC, Canada, 2008

[23] WU, Di - ZHANG, Fan - AO, Naiyong - WANG, Fang - LIU, Xiaoguang - WANG, Gang, *A Batched GPU Algorithm for Set Intersection*, Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks , Kaohsiung, Taiwan, R.O.C., 2009

[24] AO, Naiyong - ZHANG, Fan - WU, Di - STONES, Douglas S. - WANG, Gang - LIU, Xiaoguang - LIU, Jing - LIN, Sheng, *Efficient parallel lists intersection and index compression algorithms using graphics processing units*, Proceedings of the VLDB Endowment, 2011

[25] SENGUPTA, S. - HARRIS, M. - GARLAND, M., *Efficient parallel scan algorithms for GPUs*, http://mgarland.org/files/papers/nvr-2008-004.pdf, December 2008

[26] PERL, Yehoshua - ITAI, Alon - AVNI, Haim, *Interpolation search - a log log N search*, Communications of the ACM, 1978

[27] KOUBEK, Václav, *Datové struktury*, MFF UK, 2004

[28] PAGH, Rasmus - RODLER, Flemming Friche, Cuckoo Hashing, 2001

[29] BLOOM, Burton Howard, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, 1970

[30] KRULIŠ, Martin, YAGHOB, Jakub, *Revision of Relational Joins for Multi-Core And Many-Core Architectures*, Proceedings of the Dateso 2011 Workshop on DAtabases, TExts, Specifications and Objects, Písek, Czech Republic, 2011

# Appendix A: Results

| Size | std::sort | tbb:: parallel_sort | QSSS | QSBS | BSB | BSV | BSL |
|---|---|---|---|---|---|---|---|
| **4k** | 241.2 ± 2.3 µs | 93 ± 36 µs | 2120 ± 430 µs | 2140 ± 300 µs | 820 ± 410 µs | 795 ± 41 µs | 378 ± 11 µs |
| **8k** | 529.4 ± 6.4 µs | 175.3 ± 8.4 µs | 2440 ± 290 µs | 2400 ± 330 µs | 885 ± 20 µs | 889 ± 21 µs | 431 ± 15 µs |
| **16k** | 1157 ± 18 µs | 349 ± 11 µs | 3600 ± 600 µs | 3590 ± 730 µs | 950 ± 59 µs | 955 ± 45 µs | 490 ± 17 µs |
| **32k** | 2456 ± 27 µs | 798 ± 42 µs | 5.2 ± 1.1 ms | 5140 ± 980 µs | 1126 ± 48 µs | 1123.3 ± 9.4 µs | 590 ± 11 µs |
| **64k** | 5189 ± 28 µs | 1528 ± 30 µs | 9.5 ± 2.6 ms | 9.7 ± 2.6 ms | 1343 ± 19 µs | 1334 ± 17 µs | 852 ± 11 µs |
| **128k** | 10909 ± 48 µs | 2857 ± 74 µs | 14.3 ± 3.7 ms | 13,8 ± 2.9 ms | 1883.2 ± 8.8 µs | 1875 ± 10 µs | 1380 ± 20 µs |
| **256k** | 23078 ± 79 µs | 6540 ± 490 µs | 24 ± 6.1 ms | 23.5 ± 5.5 ms | 3711 ± 12 µs | 3673 ± 11 µs | 2492 ± 24 µs |
| **512k** | 48410 ± 170 µs | 13.3 ± 1.5 ms | 52 ± 17 ms | 51 ± 17 ms | 6702 ± 24 µs | 6810 ± 28 µs | 4718 ± 43 µs |
| **1M** | 101420 ± 260 µs | 26 ± 2.5 ms | 95 ± 35 ms | 92 ± 28 ms | 13421 ± 55 µs | 13612 ± 45 µs | 9329 ± 37 µs |
| **2M** | 212880 ± 540 µs | 55.7 ± 1.4 ms | 172 ± 37 ms | 168 ± 31 ms | 27710 ± 89 µs | 28078 ± 92 µs | 19880 ± 160 µs |
| **4M** | 445.2 ± 1.4 ms | 104.3 ± 3.6 ms | 530 ± 160 ms | 500 ± 130 ms | 57030 ± 240 µs | 57820 ± 210 µs | 40820 ± 140 µs |
| **8M** | 920.6 ± 3.5 ms | 239 ± 16 ms | 700 ± 170 ms | 690 ± 190 ms | 119450 ± 260 µs | 121040 ± 180 µs | 86010 ± 230 µs |
| **16M** | 1916.5 ± 7.5 ms | 470 ± 23 ms | 1780 ± 620 ms | 1950 ± 620 ms | 252200 ± 610 µs | 255720 ± 360 µs | 182030 ± 540 µs |
| **Size** | **MSB** | **MST** | **MSCBS** | **MSCBS32** | **MISCB** | **MISCBS32** | |
| **4k** | 369 ± 17 µs | 348 ± 16 µs | 364 ± 16 µs | 334 ± 21 µs | 357 ± 12 µs | 340 ± 33 µs | |
| **8k** | 472 ± 16 µs | 368 ± 14 µs | 386 ± 16 µs | 356 ± 20 µs | 378 ± 15 µs | 353.3 ± 9.1 µs | |
| **16k** | 598 ± 47 µs | 435 ± 24 µs | 419.7 ± 8.6 µs | 418 ± 15 µs | 427 ± 11 µs | 428 ± 59 µs | |
| **32k** | 838.3 ± 9.7 µs | 610 ± 14 µs | 552 ± 10 µs | 546 ± 10 µs | 572 ± 11 µs | 570 ± 150 µs | |
| **64k** | 1468 ± 47 µs | 853.7 ± 9.2 µs | 797 ± 15 µs | 823 ± 18 µs | 797.8 ± 9.1 µs | 831 ± 13 µs | |
| **128k** | 2694 ± 20 µs | 1344.2 ± 10 µs | 1267 ± 13 µs | 1281 ± 10 µs | 1264.8 ± 9.5 µs | 1286.8 ± 9 µs | |
| **256k** | 5051 ± 28 µs | 2497.2 ± 9.7 µs | 2354 ± 12 µs | 2379 ± 11 µs | 2322 ± 11 µs | 2362 ± 10 µs | |
| **512k** | 9824 ± 32 µs | 4682 ± 26 µs | 4385 ± 35 µs | 4443 ± 40 µs | 4276 ± 29 µs | 4359 ± 23 µs | |
| **1M** | 19780 ± 150 µs | 9272 ± 44 µs | 8675 ± 43 µs | 8772 ± 37 µs | 8368 ± 34 µs | 8540 ± 37 µs | |
| **2M** | 40650 ± 100 µs | 19250 ± 120 µs | 18190 ± 110 µs | 18388 ± 99 µs | 17262 ± 100 µs | 17598 ± 95 µs | |
| **4M** | 84030 ± 170 µs | 40150 ± 170 µs | 37930 ± 160 µs | 38190 ± 270 µs | 35490 ± 160 µs | 35850 ± 400 µs | |
| **8M** | 174950 ± 210 µs | 82330 ± 480 µs | 78160 ± 210 µs | 78940 ± 340 µs | 72920 ± 250 µs | 74110 ± 170 µs | |
| **16M** | 365350 ± 720 µs | 172970 ± 430 µs | 164220 ± 410 µs | 165680 ± 480 µs | 150090 ± 400 µs | 152440 ± 380 µs | |

Table 1: Sorting time: keys-only sequences

| Size | std::sort | tbb:: parallel_sort | QSSS | QSBS | BSB | BSV | BSL |
|---|---|---|---|---|---|---|---|
| **4k** | 254 ± 13 µs | 105 ± 36 µs | 2220 ± 390 µs | 2130 ± 260 µs | 891 ± 35 µs | 903 ± 22 µs | 516 ± 23 µs |
| **8k** | 558.5 ± 6 µs | 205 ± 10 µs | 2540 ± 400 µs | 2500 ± 500 µs | 1020 ± 130 µs | 1005 ± 30 µs | 577 ± 18 µs |
| **16k** | 1209 ± 17 µs | 391 ± 13 µs | 3730 ± 680 µs | 3690 ± 550 µs | 1122 ± 25 µs | 1121 ± 22 µs | 664 ± 13 µs |
| **32k** | 2552 ± 20 µs | 885 ± 43 µs | 5.3 ± 1.2 ms | 5.3 ± 1.1 ms | 1401 ± 19 µs | 1357 ± 15 µs | 923 ± 13 µs |
| **64k** | 5411 ± 21 µs | 1688 ± 28 µs | 10.6 ± 2.6 ms | 10.5 ± 2.2 ms | 1882 ± 15 µs | 1928 ± 28 µs | 1377 ± 36 µs |
| **128k** | 11298 ± 32 µs | 3178 ± 86 µs | 15.5 ± 4.2 ms | 16 ± 3.7 ms | 3532 ± 64 µs | 3566 ± 39 µs | 2412 ± 24 µs |
| **256k** | 23926 ± 65 µs | 7130 ± 470 µs | 26.3 ± 6.3 ms | 25.1 ± 4.8 ms | 6556 ± 67 µs | 6696 ± 41 µs | 4385 ± 30 µs |
| **512k** | 50400 ± 110 µs | 14.6 ± 1.5 ms | 55 ± 18 ms | 54 ± 14 ms | 12477 ± 21 µs | 12700 ± 15 µs | 8271 ± 41 µs |
| **1M** | 105610 ± 240 µs | 29 ± 2.4 ms | 111 ± 34 ms | 107 ± 34 ms | 25009 ± 27 µs | 25396 ± 39 µs | 16400 ± 110 µs |
| **2M** | 221670 ± 530 µs | 62.5 ± 1.3 ms | 182 ± 50 ms | 179 ± 40 ms | 51811 ± 90 µs | 52690 ± 86 µs | 33810 ± 140 µs |
| **4M** | 461090 ± 960 µs | 113.4 ± 3.7 ms | 510 ± 150 ms | 520 ± 160 ms | 108670 ± 270 µs | 110520 ± 190 µs | 70620 ± 170 µs |
| **8M** | 954.4 ± 2.3 ms | 260 ± 16 ms | 780 ± 20 ms | 760 ± 220 ms | 227430 ± 630 µs | 231760 ± 280 µs | 150170 ± 290 µs |
| **16M** | 1994.8 ± 4.7 ms | 519 ± 23 ms | 2160 ± 780 ms | 2140 ± 750 ms | 478990 ± 310 µs | 485680 ± 720 µs | 315000 ± 990 µs |
| **Size** | **MSB** | **MST** | **MSCBS** | **MSCBS32** | **MISCB** | **MISCBS32** | |
| **4k** | 484 ± 28 µs | 417 ± 16 µs | 487 ± 12 µs | 427 ± 13 µs | 481.9 ± 9.8 µs | 429 ± 13 µs | |
| **8k** | 552 ± 15 µs | 454 ± 14 µs | 515 ± 12 µs | 453 ± 18 µs | 515 ± 12 µs | 454 ± 17 µs | |
| **16k** | 708 ± 12 µs | 591 ± 73 µs | 599 ± 16 µs | 569 ± 13 µs | 600 ± 11 µs | 572 ± 11 µs | |
| **32k** | 1186 ± 11 µs | 865 ± 11 µs | 858 ± 11 µs | 849.7 ± 9.4 µs | 857.6 ± 9.6 µs | 848 ± 13 µs | |
| **64k** | 2040 ± 68 µs | 1293 ± 32 µs | 1192 ± 67 µs | 1216 ± 16 µs | 1173 ± 20 µs | 1290 ± 92 µs | |
| **128k** | 3642 ± 60 µs | 2207 ± 25 µs | 2147 ± 24 µs | 2153 ± 21 µs | 2134 ± 23 µs | 2149 ± 17 µs | |
| **256k** | 6698 ± 43 µs | 3921 ± 31 µs | 3849 ± 24 µs | 3852 ± 24 µs | 3829 ± 23 µs | 3840 ± 25 µs | |
| **512k** | 12722 ± 58 µs | 7175 ± 40 µs | 7082 ± 49 µs | 7050 ± 220 µs | 6961 ± 49 µs | 6966 ± 40 µs | |
| **1M** | 25108 ± 88 µs | 13996 ± 72 µs | 13691 ± 91 µs | 13669 ± 82 µs | 13458 ± 75 µs | 13465 ± 73 µs | |
| **2M** | 50580 ± 200 µs | 28270 ± 180 µs | 27438 ± 76 µs | 27130 ± 220 µs | 26700 ± 210 µs | 26760 ± 200 µs | |
| **4M** | 103730 ± 360 µs | 57500 ± 160 µs | 56390 ± 480 µs | 56020 ± 190 µs | 53980 ± 190 µs | 54100 ± 170 µs | |
| **8M** | 213530 ± 240 µs | 116730 ± 310 µs | 114950 ± 330 µs | 114590 ± 240 µs | 109540 ± 330 µs | 109750 ± 250 µs | |
| **16M** | 441720 ± 440 µs | 240110 ± 440 µs | 236560 ± 490 µs | 235910 ± 410 µs | 223040 ± 450 µs | 223370 ± 470 µs | |

Table 2: Sorting time: key-value pair sequences

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 37.5 ± 9.7 µs | 36.15 ± 0.89 µs | 37.2 ± 1.1 µs | 39.7 ± 1.2 µs | 24.2 ± 1 µs | 18.2 ± 1.2 µs |
| **8k** | 73.5 ± 1.7 µs | 73.4 ± 1.5 µs | 75.5 ± 1.9 µs | 81.5 ± 1.9 µs | 49.3 ± 1.2 µs | 36.2 ± 1.2 µs |
| **16k** | 148.8 ± 2.1 µs | 147.9 ± 2.2 µs | 152 ± 2.4 µs | 164.2 ± 2.7 µs | 100.9 ± 3 µs | 73.8 ± 2.4 µs |
| **32k** | 297.8 ± 2.9 µs | 293 ± 3 µs | 306.3 ± 3.3 µs | 329.9 ± 3.8 µs | 201.6 ± 3.2 µs | 152.1 ± 4.1 µs |
| **64k** | 597.4 ± 3.5 µs | 595.9 ± 3.2 µs | 612.7 ± 3.7 µs | 660.2 ± 5.7 µs | 404.2 ± 6.7 µs | 295.6 ± 6.4 µs |
| **128k** | 1192 ± 1.6 µs | 1192.9 ± 1.7 µs | 1227 ± 3.1 µs | 1317 ± 7 µs | 811 ± 14 µs | 593 ± 12 µs |
| **256k** | 2354.8 ± 1.9 µs | 2384.3 ± 3.2 µs | 2435.8 ± 4.6 µs | 2639 ± 15 µs | 1610 ± 26 µs | 1188 ± 28 µs |
| **512k** | 4705 ± 15 µs | 4775 ± 4.7 µs | 4915.5 ± 9.9 µs | 5279 ± 34 µs | 3227 ± 56 µs | 2378 ± 61 µs |
| **1M** | 9583.6 ± 8 µs | 9586.1 ± 7.8 µs | 9908 ± 20 µs | 10665 ± 37 µs | 6596 ± 68 µs | 4836 ± 76 µs |
| **2M** | 19050 ± 11 µs | 19031.8 ± 9.2 µs | 19864 ± 16 µs | 21312 ± 68 µs | 13100 ± 100 µs | 9650 ± 120 µs |
| **4M** | 38520 ± 24 µs | 38562 ± 21 µs | 39751 ± 43 µs | 42700 ± 120 µs | 26200 ± 200 µs | 19320 ± 210 µs |
| **8M** | 77071 ± 14 µs | 77131 ± 24 µs | 79900 ± 340 µs | 87250 ± 160 µs | 55830 ± 130 µs | 42962 ± 37 µs |
| **16M** | 154080 ± 29 µs | 153740 ± 750 µs | 159590 ± 580 µs | 174020 ± 310 µs | 111470 ± 240 µs | 85348 ± 49 µs |

Table 3: Intersection time: CPU merge-join

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 181 ± 28 µs | 186 ± 18 µs | 318 ± 15 µs | 321 ± 25 µs | 320 ± 16 µs | 324 ± 23 µs |
| **8k** | 322 ± 21 µs | 326 ± 23 µs | 322 ± 20 µs | 325 ± 17 µs | 349 ± 25 µs | 348 ± 22 µs |
| **16k** | 364 ± 16 µs | 363 ± 12 µs | 364 ± 24 µs | 388 ± 20 µs | 397 ± 21 µs | 399 ± 20 µs |
| **32k** | 468 ± 16 µs | 467 ± 14 µs | 472 ± 16 µs | 504 ± 21 µs | 547 ± 18 µs | 540 ± 22 µs |
| **64k** | 595 ± 11 µs | 593 ± 27 µs | 613.5 ± 9.4 µs | 610 ± 36 µs | 675 ± 54 µs | 768 ± 35 µs |
| **128k** | 1062 ± 22 µs | 939 ± 97 µs | 1102 ± 17 µs | 1241 ± 22 µs | 1123 ± 85 µs | 1120 ± 92 µs |
| **256k** | 1570 ± 160 µs | 1789 ± 22 µs | 1867 ± 25 µs | 1844 ± 93 µs | 2270 ± 130 µs | 2060 ± 51 µs |
| **512k** | 2330 ± 100 µs | 2707 ± 27 µs | 2700 ± 110 µs | 2937 ± 72 µs | 3395 ± 61 µs | 3397 ± 57 µs |
| **1M** | 4247 ± 67 µs | 4263 ± 98 µs | 4533 ± 89 µs | 5342 ± 41 µs | 6295 ± 31 µs | 6561 ± 23 µs |
| **2M** | 7903 ± 82 µs | 7901 ± 73 µs | 8460 ± 110 µs | 10121 ± 44 µs | 11700 ± 150 µs | 12090 ± 87 µs |
| **4M** | 14866 ± 66 µs | 14860 ± 110 µs | 15711 ± 48 µs | 19180 ± 120 µs | 22650 ± 140 µs | 23130 ± 180 µs |
| **8M** | 28840 ± 120 µs | 28894 ± 99 µs | 30724 ± 52 µs | 36820 ± 190 µs | 42960 ± 250 µs | 44330 ± 120 µs |
| **16M** | 56820 ± 140 µs | 56816 ± 94 µs | 60460 ± 200 µs | 72640 ± 210 µs | 84710 ± 240 µs | 87670 ± 250 µs |

Table 4: Host-GPU memory transfer only

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 201.7 ± 8.7 µs | 217 ± 16 µs | 414 ± 30 µs | 423 ± 36 µs | 449 ± 39 µs | 469 ± 20 µs |
| **8k** | 381 ± 21 µs | 386 ± 23 µs | 424 ± 31 µs | 455 ± 33 µs | 460 ± 22 µs | 467 ± 20 µs |
| **16k** | 433 ± 24 µs | 457 ± 15 µs | 464 ± 31 µs | 473 ± 35 µs | 510 ± 44 µs | 511 ± 20 µs |
| **32k** | 525 ± 16 µs | 579 ± 30 µs | 585 ± 26 µs | 658 ± 13 µs | 713 ± 13 µs | 724 ± 14 µs |
| **64k** | 699 ± 15 µs | 729 ± 30 µs | 765 ± 11 µs | 873 ± 33 µs | 1012 ± 14 µs | 967 ± 25 µs |
| **128k** | 1213 ± 34 µs | 1249 ± 19 µs | 1308 ± 23 µs | 1507 ± 17 µs | 1703 ± 18 µs | 1760.2 ± 8.9 µs |
| **256k** | 2205 ± 13 µs | 2255 ± 17 µs | 2364 ± 11 µs | 2760 ± 16 µs | 3145 ± 17 µs | 3200 ± 110 µs |
| **512k** | 3180 ± 120 µs | 3345 ± 16 µs | 3542 ± 26 µs | 4312 ± 12 µs | 4846 ± 98 µs | 5046 ± 99 µs |
| **1M** | 5219 ± 22 µs | 5220 ± 26 µs | 5519 ± 21 µs | 7280 ± 210 µs | 8660 ± 240 µs | 9070 ± 250 µs |
| **2M** | 9480 ± 140 µs | 9530 ± 45 µs | 9910 ± 300 µs | 13090 ± 410 µs | 15810 ± 620 µs | 16640 ± 340 µs |
| **4M** | 17908 ± 77 µs | 17850 ± 160 µs | 19070 ± 120 µs | 24920 ± 190 µs | 30700 ± 260 µs | 32000 ± 360 µs |
| **8M** | 34790 ± 160 µs | 35020 ± 160 µs | 38320 ± 550 µs | 50650 ± 460 µs | 62550 ± 160 µs | 64240 ± 930 µs |
| **16M** | 69040 ± 130 µs | 69160 ± 130 µs | 76190 ± 300 µs | 98.7 ± 1.8 ms | 121 ± 2.2 ms | 126670 ± 540 µs |

Table 5: Intersection time: Binary search (BSS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 200.7 ± 6.9 µs | 217 ± 16 µs | 391 ± 24 µs | 432 ± 50 µs | 443 ± 43 µs | 454 ± 38 µs |
| **8k** | 383 ± 23 µs | 385 ± 21 µs | 427 ± 35 µs | 461 ± 49 µs | 474 ± 12 µs | 469 ± 18 µs |
| **16k** | 425 ± 14 µs | 471 ± 24 µs | 470 ± 44 µs | 472 ± 28 µs | 508 ± 50 µs | 513 ± 22 µs |
| **32k** | 521 ± 17 µs | 570 ± 27 µs | 589 ± 25 µs | 661 ± 13 µs | 715 ± 14 µs | 724.9 ± 9.6 µs |
| **64k** | 701 ± 15 µs | 736 ± 12 µs | 769.3 ± 9.7 µs | 863 ± 30 µs | 1018 ± 12 µs | 978 ± 26 µs |
| **128k** | 1235 ± 31 µs | 1254 ± 18 µs | 1314 ± 17 µs | 1522 ± 19 µs | 1715 ± 17 µs | 1761 ± 12 µs |
| **256k** | 2215 ± 18 µs | 2265 ± 14 µs | 2375 ± 14 µs | 2758.3 ± 5.1 µs | 3161 ± 13 µs | 3244 ± 12 µs |
| **512k** | 3321 ± 27 µs | 3368 ± 17 µs | 3565 ± 29 µs | 4346 ± 14 µs | 4901 ± 26 µs | 5063 ± 16 µs |
| **1M** | 5321 ± 36 µs | 5450 ± 52 µs | 6097 ± 76 µs | 7425 ± 21 µs | 8806 ± 40 µs | 9159 ± 18 µs |
| **2M** | 9830 ± 33 µs | 9530 ± 130 µs | 10096 ± 22 µs | 13150 ± 370 µs | 16720 ± 110 µs | 16740 ± 340 µs |
| **4M** | 18548 ± 38 µs | 18390 ± 150 µs | 20175 ± 79 µs | 25600 ± 260 µs | 31180 ± 300 µs | 32720 ± 150 µs |
| **8M** | 36350 ± 160 µs | 36780 ± 130 µs | 40390 ± 130 µs | 54490 ± 160 µs | 63530 ± 380 µs | 68.1 ± 1.7 ms |
| **16M** | 74380 ± 340 µs | 75680 ± 150 µs | 84230 ± 340 µs | 105.8 ± 1.2 ms | 127130 ± 770 µs | 132.4 ± 1.5 ms |

Table 6: Intersection time: Binary search with look-up (BSLS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 218 ± 19 µs | 217 ± 15 µs | 394 ± 26 µs | 416 ± 29 µs | 429 ± 26 µs | 451 ± 37 µs |
| **8k** | 380 ± 18 µs | 372 ± 19 µs | 431 ± 49 µs | 458 ± 12 µs | 471 ± 32 µs | 470 ± 18 µs |
| **16k** | 422 ± 15 µs | 467 ± 55 µs | 445 ± 25 µs | 470 ± 17 µs | 507 ± 44 µs | 516 ± 24 µs |
| **32k** | 514 ± 17 µs | 557 ± 28 µs | 590 ± 31 µs | 652 ± 11 µs | 706 ± 16 µs | 715.8 ± 9.3 µs |
| **64k** | 670 ± 24 µs | 725 ± 14 µs | 738 ± 21 µs | 875 ± 33 µs | 993 ± 43 µs | 997 ± 41 µs |
| **128k** | 1192 ± 18 µs | 1223 ± 17 µs | 1281 ± 18 µs | 1481 ± 25 µs | 1674 ± 21 µs | 1719 ± 15 µs |
| **256k** | 2153 ± 19 µs | 2202 ± 20 µs | 2310 ± 13 µs | 2676 ± 15 µs | 3072 ± 12 µs | 3163 ± 17 µs |
| **512k** | 3169 ± 16 µs | 3218 ± 17 µs | 3418 ± 21 µs | 4165 ± 16 µs | 4703 ± 29 µs | 4901 ± 23 µs |
| **1M** | 4930 ± 150 µs | 5011 ± 90 µs | 5680 ± 210 µs | 7061 ± 31 µs | 8352 ± 26 µs | 8720 ± 120 µs |
| **2M** | 8830 ± 130 µs | 8770 ± 84 µs | 9298 ± 58 µs | 12680 ± 390 µs | 15400 ± 300 µs | 16300 ± 200 µs |
| **4M** | 16730 ± 170 µs | 16550 ± 160 µs | 17830 ± 280 µs | 23420 ± 230 µs | 28850 ± 130 µs | 30120 ± 200 µs |
| **8M** | 32080 ± 140 µs | 32080 ± 180 µs | 35490 ± 530 µs | 47390 ± 500 µs | 58.9 ± 1.7 ms | 62300 ± 780 µs |
| **16M** | 63345 ± 83 µs | 63640 ± 130 µs | 70470 ± 250 µs | 93110 ± 600 µs | 113.5 ± 1.9 ms | 117900 ± 690 µs |

Table 7: Intersection time: Interpolation search (ISS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 218 ± 16 µs | 248 ± 42 µs | 408 ± 33 µs | 450 ± 38 µs | 464 ± 19 µs | 458 ± 42 µs |
| **8k** | 392 ± 20 µs | 386 ± 22 µs | 425 ± 34 µs | 460 ± 14 µs | 459 ± 25 µs | 470.6 ± 7.3 µs |
| **16k** | 420 ± 24 µs | 455 ± 35 µs | 466 ± 37 µs | 472 ± 30 µs | 517 ± 43 µs | 502 ± 25 µs |
| **32k** | 530 ± 15 µs | 570 ± 28 µs | 604 ± 17 µs | 658 ± 11 µs | 711 ± 11 µs | 722.3 ± 9.6 µs |
| **64k** | 700 ± 15 µs | 736 ± 14 µs | 773 ± 18 µs | 872 ± 36 µs | 975 ± 29 µs | 998 ± 50 µs |
| **128k** | 1209 ± 23 µs | 1244 ± 16 µs | 1306 ± 25 µs | 1502 ± 20 µs | 1695 ± 17 µs | 1744 ± 28 µs |
| **256k** | 2193 ± 22 µs | 2236 ± 29 µs | 2346 ± 14 µs | 2718 ± 14 µs | 3111 ± 13 µs | 3190 ± 14 µs |
| **512k** | 3271 ± 13 µs | 3311 ± 14 µs | 3506 ± 21 µs | 4253 ± 18 µs | 4789 ± 28 µs | 4991 ± 36 µs |
| **1M** | 5488 ± 37 µs | 5541 ± 29 µs | 5943 ± 19 µs | 7213 ± 32 µs | 8544 ± 27 µs | 8893 ± 41 µs |
| **2M** | 9707 ± 54 µs | 9370 ± 240 µs | 10050 ± 170 µs | 13150 ± 340 µs | 15830 ± 360 µs | 16400 ± 200 µs |
| **4M** | 18146 ± 72 µs | 17990 ± 170 µs | 19140 ± 210 µs | 24540 ± 210 µs | 29890 ± 190 µs | 31080 ± 190 µs |
| **8M** | 35550 ± 120 µs | 35920 ± 62 µs | 38600 ± 490 µs | 51840 ± 550 µs | 63010 ± 230 µs | 65490 ± 630 µs |
| **16M** | 73620 ± 340 µs | 74160 ± 160 µs | 81540 ± 370 µs | 103090 ± 590 µs | 122 ± 2.2 ms | 127210 ± 550 µs |

Table 8: Intersection time: Interpolation search with look-up (ISLS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 226 ± 23 µs | 272 ± 34 µs | 404 ± 34 µs | 448 ± 19 µs | 432 ± 29 µs | 446 ± 38 µs |
| **8k** | 388 ± 18 µs | 389 ± 17 µs | 413 ± 21 µs | 457 ± 13 µs | 463 ± 32 µs | 473 ± 11 µs |
| **16k** | 426 ± 13 µs | 445 ± 30 µs | 464 ± 35 µs | 476 ± 25 µs | 503 ± 25 µs | 486 ± 20 µs |
| **32k** | 529 ± 15 µs | 570 ± 26 µs | 604 ± 16 µs | 661 ± 19 µs | 716 ± 10 µs | 725 ± 15 µs |
| **64k** | 696 ± 21 µs | 740 ± 11 µs | 765 ± 22 µs | 879 ± 23 µs | 975 ± 28 µs | 985 ± 29 µs |
| **128k** | 1213 ± 17 µs | 1263 ± 28 µs | 1292 ± 22 µs | 1516 ± 17 µs | 1705 ± 27 µs | 1747 ± 27 µs |
| **256k** | 2235 ± 17 µs | 2284.8 ± 8.4 µs | 2386 ± 11 µs | 2761 ± 17 µs | 3146 ± 16 µs | 3224 ± 12 µs |
| **512k** | 3337 ± 30 µs | 3374 ± 18 µs | 3583 ± 22 µs | 4318 ± 13 µs | 4845 ± 68 µs | 5033 ± 30 µs |
| **1M** | 5585 ± 30 µs | 5647 ± 34 µs | 6061 ± 19 µs | 7318 ± 27 µs | 8635 ± 29 µs | 8985 ± 90 µs |
| **2M** | 9590 ± 200 µs | 9265 ± 86 µs | 10120 ± 240 µs | 13160 ± 310 µs | 15770 ± 340 µs | 16350 ± 130 µs |
| **4M** | 17950 ± 160 µs | 17880 ± 170 µs | 19170 ± 200 µs | 24500 ± 220 µs | 29940 ± 220 µs | 31076 ± 81 µs |
| **8M** | 34610 ± 160 µs | 34560 ± 150 µs | 37820 ± 490 µs | 49.2 ± 1.1 ms | 62100 ± 300 µs | 63200 ± 440 µs |
| **16M** | 68462 ± 83 µs | 69130 ± 180 µs | 75370 ± 220 µs | 96.1 ± 1.4 ms | 116600 ± 610 µs | 121.1 ± 1.3 ms |

Table 9: Intersection time: Generalized quadratic search (GQSS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 307 ± 28 µs | 519 ± 24 µs | 581 ± 61 µs | 575 ± 40 µs | 624 ± 48 µs | 625 ± 61 µs |
| **8k** | 541 ± 28 µs | 556 ± 35 µs | 568 ± 43 µs | 630 ± 220 µs | 617 ± 30 µs | 655 ± 16 µs |
| **16k** | 584 ± 25 µs | 618 ± 51 µs | 640 ± 44 µs | 637 ± 28 µs | 696 ± 76 µs | 698 ± 34 µs |
| **32k** | 717 ± 22 µs | 729 ± 25 µs | 784 ± 61 µs | 857 ± 14 µs | 912 ± 37 µs | 921.9 ± 9 µs |
| **64k** | 891 ± 56 µs | 942 ± 31 µs | 942 ± 39 µs | 1040 ± 36 µs | 1137 ± 46 µs | 1182 ± 65 µs |
| **128k** | 1340 ± 23 µs | 1458 ± 31 µs | 1461 ± 18 µs | 1745 ± 16 µs | 1920 ± 18 µs | 1955 ± 17 µs |
| **256k** | 2450 ± 21 µs | 2493 ± 15 µs | 2608 ± 18 µs | 3029 ± 27 µs | 3415 ± 21 µs | 3471 ± 19 µs |
| **512k** | 3470 ± 120 µs | 3581 ± 26 µs | 3790 ± 22 µs | 4624 ± 20 µs | 5172 ± 21 µs | 5318 ± 22 µs |
| **1M** | 5733 ± 45 µs | 5764 ± 30 µs | 6259 ± 26 µs | 7720 ± 31 µs | 9141 ± 26 µs | 9494 ± 77 µs |
| **2M** | 9330 ± 120 µs | 9300 ± 190 µs | 9940 ± 110 µs | 13270 ± 600 µs | 16740 ± 180 µs | 16950 ± 570 µs |
| **4M** | 17540 ± 260 µs | 17250 ± 250 µs | 19090 ± 340 µs | 26360 ± 100 µs | 31680 ± 350 µs | 32050 ± 450 µs |
| **8M** | 34260 ± 250 µs | 34050 ± 390 µs | 37730 ± 700 µs | 51530 ± 700 µs | 62940 ± 450 µs | 64560 ± 950 µs |
| **16M** | 67460 ± 590 µs | 67830 ± 480 µs | 75540 ± 280 µs | 98.8 ± 1.7 ms | 121.6 ± 2.2 ms | 123420 ± 220 µs |

Table 10: Intersection time: Local search with jobs created on CPU (LSJCCPUS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 315 ± 34 µs | 534 ± 27 µs | 588 ± 39 µs | 581 ± 30 µs | 637 ± 73 µs | 617 ± 37 µs |
| **8k** | 555 ± 26 µs | 562 ± 26 µs | 614 ± 65 µs | 624 ± 49 µs | 641 ± 27 µs | 671 ± 52 µs |
| **16k** | 629 ± 31 µs | 655 ± 22 µs | 672 ± 59 µs | 676 ± 52 µs | 712 ± 27 µs | 730 ± 36 µs |
| **32k** | 757 ± 36 µs | 781 ± 53 µs | 833 ± 39 µs | 902 ± 40 µs | 954 ± 41 µs | 992 ± 16 µs |
| **64k** | 938 ± 18 µs | 985 ± 46 µs | 1017 ± 13 µs | 1155 ± 13 µs | 1233 ± 32 µs | 1285 ± 30 µs |
| **128k** | 1495 ± 57 µs | 1585 ± 28 µs | 1670 ± 70 µs | 1877 ± 24 µs | 2084 ± 41 µs | 2206 ± 17 µs |
| **256k** | 2658 ± 23 µs | 2733 ± 31 µs | 2846 ± 18 µs | 3323 ± 14 µs | 3810 ± 24 µs | 3950 ± 14 µs |
| **512k** | 3733 ± 29 µs | 4013 ± 25 µs | 4249 ± 16 µs | 5149 ± 17 µs | 5943 ± 33 µs | 6238 ± 26 µs |
| **1M** | 6634 ± 37 µs | 6671 ± 42 µs | 7144 ± 30 µs | 8786 ± 42 µs | 10477 ± 77 µs | 11159 ± 95 µs |
| **2M** | 11010 ± 150 µs | 10930 ± 100 µs | 11750 ± 180 µs | 15310 ± 150 µs | 18760 ± 220 µs | 19990 ± 430 µs |
| **4M** | 21233 ± 73 µs | 21170 ± 160 µs | 22410 ± 230 µs | 29840 ± 240 µs | 35960 ± 200 µs | 39030 ± 240 µs |
| **8M** | 41340 ± 200 µs | 41380 ± 170 µs | 45000 ± 490 µs | 59790 ± 660 µs | 73150 ± 320 µs | 78.7 ± 1.4 ms |
| **16M** | 82380 ± 170 µs | 82070 ± 390 µs | 89040 ± 560 µs | 116420 ± 970 µs | 142.8 ± 1.7 ms | 152.6 ± 1 ms |

Table 11: Intersection time: Local search with jobs created on GPU (LSJCGPUS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 211 ± 90 µs | 192 ± 11 µs | 199 ± 13 µs | 216 ± 13 µs | 200 ± 12 µs | 189.1 ± 9.5 µs |
| **8k** | 355 ± 18 µs | 358 ± 19 µs | 367 ± 19 µs | 399 ± 14 µs | 373 ± 15 µs | 351 ± 19 µs |
| **16k** | 691 ± 16 µs | 726 ± 16 µs | 718 ± 13 µs | 795 ± 14 µs | 722 ± 22 µs | 676 ± 18 µs |
| **32k** | 1441 ± 20 µs | 1374 ± 29 µs | 1664 ± 28 µs | 1493 ± 44 µs | 1341 ± 23 µs | 1274 ± 14 µs |
| **64k** | 3029 ± 86 µs | 2678 ± 35 µs | 2878 ± 41 µs | 2953 ± 29 µs | 2762 ± 48 µs | 2634 ± 53 µs |
| **128k** | 5660 ± 160 µs | 6100 ± 170 µs | 5690 ± 190 µs | 6082 ± 73 µs | 5646 ± 96 µs | 5260 ± 130 µs |
| **256k** | 11570 ± 600 µs | 13360 ± 470 µs | 12560 ± 250 µs | 13240 ± 510 µs | 11090 ± 210 µs | 10960 ± 570 µs |
| **512k** | 23570 ± 820 µs | 22380 ± 950 µs | 23190 ± 500 µs | 26610 ± 420 µs | 23240 ± 460 µs | 23140 ± 350 µs |
| **1M** | 46.3 ± 1.0 ms | 49.2 ± 2.7 ms | 49.4 ± 1.7 ms | 50400 ± 830 µs | 47210 ± 890 µs | 45.6 ± 1.3 ms |
| **2M** | 99.9 ± 4.8 ms | 94.7 ± 3.0 ms | 103.6 ± 5.9 ms | 105.5 ± 4.2 ms | 98.1 ± 3.1 ms | 93.1 ± 2.2 ms |
| **4M** | 223 ± 4.6 ms | 212.5 ± 7.1 ms | 206.6 ± 4.4 ms | 239.2 ± 4.8 ms | 196.9 ± 3.5 ms | 209.4 ± 6.2 ms |
| **8M** | 419 ± 14 ms | 393 ± 11 ms | 460 ± 10 ms | 486 ± 13 ms | 417 ± 25 ms | 387.2 ± 6.2 ms |
| **16M** | 894 ± 30 ms | 893 ± 25 ms | 872 ± 20 ms | 986 ± 17 ms | 876 ± 35 ms | 831 ± 31 ms |

Table 12: Intersection times: CPU tbb::sort + merge-join

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 72.8 ± 2.8 ms | 74.2 ± 2.7 ms | 73.7 ± 3.1 ms | 73.7 ± 3.1 ms | 73.1 ± 3.0 ms | 72.9 ± 2.9 ms |
| **8k** | 73,8 ± 3.0 ms | 73.4 ± 3.1 ms | 73.8 ± 3.5 ms | 73.9 ± 3.3 ms | 73.8 ± 3.2 ms | 73.7 ± 3.6 ms |
| **16k** | 74.1 ± 3.4 ms | 74.7 ± 2.9 ms | 74.2 ± 2.9 ms | 73.8 ± 3.6 ms | 74.3 ± 3.4 ms | 74.5 ± 3.4 ms |
| **32k** | 74.4 ± 3.4 ms | 75.3 ± 3.0 ms | 74.5 ± 3.4 ms | 74.1 ± 2.8 ms | 73.4 ± 2.7 ms | 73.3 ± 2.9 ms |
| **64k** | 76.0 ± 2.7 ms | 76.0 ± 2.5 ms | 75.9 ± 2.6 ms | 75.6 ± 3.0 ms | 75.2 ± 2.5 ms | 75.2 ± 2.8 ms |
| **128k** | 78.1 ± 2.5 ms | 78.1 ± 2.7 ms | 78.3 ± 3.0 ms | 78.1 ± 2.6 ms | 77.5 ± 3.0 ms | 77.3 ± 2.5 ms |
| **256k** | 83.6 ± 2.3 ms | 83.2 ± 3.1 ms | 83.0 ± 2.3 ms | 84.1 ± 2.7 ms | 84.4 ± 2.3 ms | 83.8 ± 2.5 ms |
| **512k** | 92.9 ± 3.1 ms | 93.2 ± 2.8 ms | 92.6 ± 2.5 ms | 94.1 ± 3.3 ms | 94.2 ± 2.7 ms | 95.1 ± 2.4 ms |
| **1M** | 113.1 ± 3.0 ms | 112.9 ± 2.6 ms | 113.1 ± 2.7 ms | 114110 ± 750 µs | 114410 ± 700 µs | 114460 ± 870 µs |
| **2M** | 134270 ± 570 µs | 134380 ± 530 µs | 134970 ± 550 µs | 136860 ± 570 µs | 137180 ± 440 µs | 137460 ± 520 µs |
| **4M** | 167270 ± 410 µs | 167450 ± 410 µs | 168230 ± 460 µs | 171890 ± 430 µs | 172670 ± 700 µs | 172670 ± 470 µs |
| **8M** | 221810 ± 760 µs | 221970 ± 800 µs | 224000 ± 780 µs | 231580 ± 970 µs | 233900 ± 930 µs | 234480 ± 860 µs |
| **16M** | 325.8 ± 1.7 ms | 325.5 ± 1.4 ms | 328.5 ± 1.2 ms | 345.4 ± 1.7 ms | 348.5 ± 1.4 ms | 351.2 ± 1.9 ms |

Table 13: Intersection time: CPU two-pass bucketing

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 178 ± 23 µs | 321 ± 23 µs | 317 ± 23 µs | 321 ± 26 µs | 326 ± 21 µs | 325 ± 25 µs |
| **8k** | 340 ± 14 µs | 330 ± 28 µs | 325 ± 24 µs | 342 ± 29 µs | 363 ± 23 µs | 361 ± 23 µs |
| **16k** | 368 ± 11 µs | 363 ± 19 µs | 369 ± 11 µs | 387 ± 13 µs | 406 ± 11 µs | 399 ± 28 µs |
| **32k** | 466 ± 11 µs | 466 ± 10 µs | 468 ± 19 µs | 495 ± 18 µs | 537 ± 20 µs | 548 ± 29 µs |
| **64k** | 601 ± 14 µs | 592 ± 34 µs | 619 ± 10 µs | 540 ± 56 µs | 756 ± 27 µs | 616 ± 72 µs |
| **128k** | 1060 ± 18 µs | 1080 ± 36 µs | 1108 ± 22 µs | 1247 ± 16 µs | 1382 ± 31 µs | 1230 ± 120 µs |
| **256k** | 1797 ± 25 µs | 1790 ± 25 µs | 1874 ± 32 µs | 1860 ± 140 µs | 2022 ± 62 µs | 2079 ± 95 µs |
| **512k** | 2706 ± 23 µs | 2523 ± 100 µs | 2853 ± 18 µs | 3100 ± 100 µs | 3423 ± 60 µs | 3554 ± 66 µs |
| **1M** | 4243 ± 70 µs | 4334 ± 90 µs | 4566 ± 95 µs | 5419 ± 44 µs | 6323 ± 65 µs | 6604 ± 33 µs |
| **2M** | 7960 ± 150 µs | 8009 ± 64 µs | 8539 ± 99 µs | 10140 ± 100 µs | 11972 ± 91 µs | 12240 ± 120 µs |
| **4M** | 14982 ± 84 µs | 15112 ± 94 µs | 15982 ± 42 µs | 19468 ± 53 µs | 22490 ± 170 µs | 23540 ± 110 µs |
| **8M** | 28941 ± 79 µs | 28945 ± 77 µs | 30747 ± 88 µs | 36840 ± 170 µs | 43180 ± 230 µs | 44670 ± 190 µs |
| **16M** | 56685 ± 72 µs | 56700 ± 120 µs | 60210 ± 130 µs | 72570 ± 160 µs | 85080 ± 330 µs | 88200 ± 270 µs |

Table 14: Memory transfers only

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 250 ± 26 µs | 458 ± 31 µs | 482 ± 34 µs | 480 ± 17 µs | 478 ± 29 µs | 510 ± 39 µs |
| **8k** | 473 ± 24 µs | 468 ± 19 µs | 485 ± 34 µs | 520 ± 41 µs | 550 ± 12 µs | 524 ± 26 µs |
| **16k** | 486 ± 13 µs | 530 ± 29 µs | 543 ± 38 µs | 556 ± 24 µs | 623 ± 27 µs | 623 ± 33 µs |
| **32k** | 649 ± 13 µs | 700 ± 20 µs | 681 ± 46 µs | 750 ± 29 µs | 815 ± 36 µs | 811 ± 27 µs |
| **64k** | 1065 ± 27 µs | 1057 ± 31 µs | 1124 ± 63 µs | 1110 ± 130 µs | 1343 ± 54 µs | 1280 ± 150 µs |
| **128k** | 1715 ± 31 µs | 1747 ± 18 µs | 1774 ± 20 µs | 1976 ± 57 µs | 2134 ± 16 µs | 2226 ± 47 µs |
| **256k** | 2810 ± 29 µs | 2840 ± 26 µs | 2932 ± 15 µs | 3237 ± 18 µs | 3605 ± 18 µs | 3752 ± 79 µs |
| **512k** | 4520 ± 32 µs | 4522 ± 54 µs | 4724 ± 30 µs | 5416 ± 31 µs | 5730 ± 120 µs | 5960 ± 77 µs |
| **1M** | 7532 ± 53 µs | 7950 ± 150 µs | 8360 ± 120 µs | 9340 ± 240 µs | 10390 ± 410 µs | 10810 ± 280 µs |
| **2M** | 14180 ± 210 µs | 14230 ± 180 µs | 14780 ± 190 µs | 17160 ± 430 µs | 19670 ± 230 µs | 20360 ± 170 µs |
| **4M** | 27540 ± 140 µs | 27800 ± 130 µs | 28730 ± 180 µs | 33650 ± 250 µs | 37810 ± 270 µs | 38640 ± 210 µs |
| **8M** | 54120 ± 110 µs | 54170 ± 160 µs | 57150 ± 230 µs | 66680 ± 400 µs | 74280 ± 390 µs | 77.0 ± 1.3 ms |
| **16M** | 107080 ± 300 µs | 107140 ± 300 µs | 112540 ± 180 µs | 129.9 ± 1.6 ms | 148100 ± 190 µs | 151340 ± 910 µs |

Table 15: Intersection time: Linear hashing with buffered output (LHB)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 247 ± 11 µs | 439 ± 19 µs | 479 ± 29 µs | 480 ± 21 µs | 483 ± 21 µs | 481 ± 25 µs |
| **8k** | 460 ± 18 µs | 450 ± 12 µs | 483 ± 14 µs | 515 ± 27 µs | 555 ± 20 µs | 525 ± 21 µs |
| **16k** | 483 ± 13 µs | 494 ± 13 µs | 527 ± 23 µs | 573 ± 35 µs | 608 ± 28 µs | 560 ± 16 µs |
| **32k** | 651 ± 27 µs | 668 ± 26 µs | 677 ± 26 µs | 734 ± 46 µs | 812 ± 46 µs | 821 ± 24 µs |
| **64k** | 1023 ± 23 µs | 1090 ± 19 µs | 1114 ± 23 µs | 1220 ± 28 µs | 1346 ± 22 µs | 1346 ± 36 µs |
| **128k** | 1651 ± 12 µs | 1708 ± 37 µs | 1753 ± 37 µs | 1948 ± 20 µs | 2127 ± 15 µs | 2234 ± 38 µs |
| **256k** | 2802 ± 29 µs | 2799 ± 17 µs | 2906 ± 19 µs | 3281 ± 42 µs | 3637 ± 82 µs | 3701 ± 34 µs |
| **512k** | 4436 ± 40 µs | 4453 ± 19 µs | 4648 ± 22 µs | 5312 ± 24 µs | 5747 ± 25 µs | 5975 ± 21 µs |
| **1M** | 7159 ± 98 µs | 7340 ± 120 µs | 7900 ± 340 µs | 9328 ± 72 µs | 10510 ± 100 µs | 10820 ± 200 µs |
| **2M** | 13920 ± 130 µs | 14010 ± 160 µs | 14460 ± 140 µs | 17020 ± 300 µs | 19540 ± 170 µs | 20310 ± 170 µs |
| **4M** | 26899 ± 53 µs | 27020 ± 300 µs | 28110 ± 200 µs | 33050 ± 190 µs | 37660 ± 240 µs | 38640 ± 190 µs |
| **8M** | 52730 ± 120 µs | 52790 ± 170 µs | 55910 ± 310 µs | 66070 ± 220 µs | 74270 ± 290 µs | 76.7 ± 1.2 ms |
| **16M** | 104260 ± 280 µs | 104390 ± 260 µs | 109730 ± 320 µs | 128.5 ± 1.6 ms | 147750 ± 160 µs | 151350 ± 950 µs |

Table 16: Intersection time: Linear hashing with sorted keys (LHS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 279 ± 22 µs | 492 ± 24 µs | 540 ± 32 µs | 560 ± 41 µs | 552 ± 29 µs | 570 ± 39 µs |
| **8k** | 516 ± 27 µs | 525 ± 22 µs | 548 ± 29 µs | 584 ± 27 µs | 637 ± 18 µs | 615 ± 44 µs |
| **16k** | 562 ± 32 µs | 599 ± 29 µs | 608 ± 31 µs | 637 ± 33 µs | 658 ± 30 µs | 629 ± 25 µs |
| **32k** | 986 ± 23 µs | 981 ± 21 µs | 980 ± 43 µs | 1034 ± 46 µs | 1129 ± 25 µs | 1114 ± 46 µs |
| **64k** | 1179 ± 40 µs | 1210 ± 36 µs | 1249 ± 58 µs | 1321 ± 44 µs | 1466 ± 20 µs | 1470 ± 27 µs |
| **128k** | 1898 ± 38 µs | 1970 ± 14 µs | 1998 ± 83 µs | 2114 ± 58 µs | 2231 ± 38 µs | 2307 ± 60 µs |
| **256k** | 3366 ± 25 µs | 3435 ± 28 µs | 3512 ± 22 µs | 3814 ± 33 µs | 4056 ± 18 µs | 4144 ± 30 µs |
| **512k** | 5390 ± 24 µs | 5473 ± 20 µs | 5608 ± 20 µs | 6145 ± 30 µs | 6414 ± 25 µs | 6560 ± 29 µs |
| **1M** | 8860 ± 230 µs | 9010 ± 220 µs | 9230 ± 250 µs | 10680 ± 280 µs | 11500 ± 180 µs | 11510 ± 410 µs |
| **2M** | 17090 ± 210 µs | 17110 ± 230 µs | 17530 ± 250 µs | 19320 ± 230 µs | 21290 ± 280 µs | 21840 ± 240 µs |
| **4M** | 33223 ± 68 µs | 33390 ± 280 µs | 33850 ± 330 µs | 37700 ± 240 µs | 40880 ± 220 µs | 41960 ± 340 µs |
| **8M** | 65300 ± 160 µs | 65530 ± 360 µs | 68010 ± 260 µs | 75660 ± 600 µs | 82290 ± 90 µs | 83120 ± 250 µs |
| **16M** | 127610 ± 450 µs | 127680 ± 430 µs | 131480 ± 580 µs | 144.9 ± 1.0 ms | 157290 ± 110 µs | 160.1 ± 1.2 ms |

Table 17: Intersection time: Cuckoo hashing to global table (CHG)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 436 ± 25 µs | 723 ± 47 µs | 761 ± 98 µs | 791 ± 69 µs | 825 ± 57 µs | 833 ± 94 µs |
| **8k** | 729 ± 88 µs | 692 ± 24 µs | 779 ± 61 µs | 813 ± 55 µs | 901 ± 48 µs | 872 ± 55 µs |
| **16k** | 776 ± 74 µs | 796 ± 49 µs | 860 ± 60 µs | 885 ± 50 µs | 943 ± 52 µs | 978 ± 80 µs |
| **32k** | 950 ± 35 µs | 999 ± 87 µs | 1003 ± 56 µs | 1111 ± 50 µs | 1264 ± 25 µs | 1231 ± 55 µs |
| **64k** | 1442 ± 49 µs | 1444 ± 54 µs | 1472 ± 28 µs | 1715 ± 27 µs | 1820 ± 57 µs | 1900 ± 100 µs |
| **128k** | 2266 ± 59 µs | 2350 ± 23 µs | 2375 ± 72 µs | 2760 ± 25 µs | 3102 ± 92 µs | 3231 ± 75 µs |
| **256k** | 3397 ± 13 µs | 3484 ± 22 µs | 3625 ± 49 µs | 4218 ± 79 µs | 4863 ± 43 µs | 4975 ± 82 µs |
| **512k** | 4845 ± 37 µs | 4881 ± 98 µs | 5180 ± 210 µs | 6480 ± 130 µs | 7387 ± 28 µs | 7691 ± 27 µs |
| **1M** | 7540 ± 420 µs | 7710 ± 420 µs | 8140 ± 460 µs | 10670 ± 120 µs | 12280 ± 520 µs | 13110 ± 330 µs |
| **2M** | 24160 ± 160 µs | 24250 ± 130 µs | 25080 ± 200 µs | 29130 ± 350 µs | 34630 ± 690 µs | 35490 ± 830 µs |
| **4M** | 43690 ± 350 µs | 43530 ± 210 µs | 45380 ± 350 µs | 55210 ± 770 µs | 63950 ± 150 µs | 66090 ± 200 µs |
| **8M** | 75110 ± 430 µs | 74650 ± 300 µs | 78650 ± 540 µs | 97680 ± 980 µs | 114180 ± 270 µs | 119.0 ± 1.0 ms |
| **16M** | 130750 ± 540 µs | 131180 ± 450 µs | 138570 ± 440 µs | 173560 ± 610 µs | 207500 ± 540 µs | 214.2 ± 1.1 ms |

Table 18: Intersection time: Cuckoo hashing to local table (CHL)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| **Size** | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 4800 ± 260 µs | 4791 ± 74 µs | 4867 ± 36 µs | 4845 ± 72 µs | 4954 ± 37 µs | 4959 ± 39 µs |
| **8k** | 4803 ± 78 µs | 4796 ± 45 µs | 4901 ± 61 µs | 4986 ± 69 µs | 4974 ± 46 µs | 4971 ± 48 µs |
| **16k** | 4818 ± 41 µs | 4887 ± 56 µs | 4963 ± 36 µs | 4952 ± 27 µs | 5000 ± 120 µs | 5069 ± 45 µs |
| **32k** | 4978 ± 31 µs | 5044 ± 54 µs | 4950 ± 170 µs | 5265 ± 44 µs | 5357 ± 59 µs | 5398 ± 84 µs |
| **64k** | 5513 ± 69 µs | 5528 ± 42 µs | 5601 ± 80 µs | 5740 ± 98 µs | 5854 ± 43 µs | 5952 ± 21 µs |
| **128k** | 6294 ± 31 µs | 6233 ± 93 µs | 6378 ± 64 µs | 6634 ± 27 µs | 6810 ± 250 µs | 6870 ± 160 µs |
| **256k** | 7965 ± 38 µs | 8028 ± 32 µs | 8090 ± 80 µs | 8410 ± 200 µs | 8993 ± 55 µs | 9099 ± 41 µs |
| **512k** | 9710 ± 490 µs | 9450 ± 490 µs | 9410 ± 340 µs | 10320 ± 510 µs | 11040 ± 570 µs | 11430 ± 490 µs |
| **1M** | 12797 ± 43 µs | 12660 ± 210 µs | 13120 ± 270 µs | 14510 ± 330 µs | 15790 ± 370 µs | 16300 ± 470 µs |
| **2M** | 20480 ± 220 µs | 20482 ± 73 µs | 21140 ± 230 µs | 23890 ± 440 µs | 26500 ± 610 µs | 26900 ± 480 µs |
| **4M** | 36140 ± 350 µs | 35920 ± 210 µs | 37230 ± 480 µs | 42560 ± 720 µs | 48550 ± 290 µs | 50200 ± 180 µs |
| **8M** | 66600 ± 270 µs | 67000 ± 420 µs | 69040 ± 500 µs | 80.5 ± 1.0 ms | 91.8 ± 1.3 ms | 94.0 ± 1.0 ms |
| **16M** | 128820 ± 390 µs | 128230 ± 460 µs | 134140 ± 930 µs | 152.5 ± 1.4 ms | 174.0 ± 1.5 ms | 180190 ± 840 µs |

Table 19: Intersection time: Indexing into large bitmap with buffered split (ILBS)

| | Ratio of elements common to both sets | | | | | |
|---|---|---|---|---|---|---|
| | **0%** | **0.01%** | **10%** | **50%** | **90%** | **100%** |
| **4k** | 4456 ± 30 µs | 4470 ± 24 µs | 4491 ± 62 µs | 4565 ± 67 µs | 4610 ± 200 µs | 4537 ± 39 µs |
| **8k** | 4477 ± 22 µs | 4467 ± 16 µs | 4537 ± 23 µs | 4554 ± 31 µs | 4608 ± 66 µs | 4613 ± 42 µs |
| **16k** | 4494 ± 24 µs | 4580 ± 34 µs | 4592 ± 26 µs | 4612 ± 65 µs | 4674 ± 28 µs | 4647 ± 17 µs |
| **32k** | 4683 ± 29 µs | 4757 ± 16 µs | 4763 ± 22 µs | 4825 ± 27 µs | 4928 ± 48 µs | 4921 ± 38 µs |
| **64k** | 4961 ± 23 µs | 5070 ± 20 µs | 5084 ± 48 µs | 5142 ± 28 µs | 5274 ± 17 µs | 5347 ± 19 µs |
| **128k** | 5685 ± 29 µs | 5719 ± 26 µs | 5833 ± 27 µs | 6025 ± 18 µs | 6248 ± 45 µs | 6243 ± 41 µs |
| **256k** | 7061 ± 38 µs | 7140 ± 30 µs | 7243 ± 60 µs | 7630 ± 18 µs | 8042 ± 20 µs | 8113 ± 70 µs |
| **512k** | 8390 ± 340 µs | 8470 ± 340 µs | 8580 ± 310 µs | 9470 ± 220 µs | 10420 ± 190 µs | 10570 ± 200 µs |
| **1M** | 11470 ± 45 µs | 11377 ± 59 µs | 11930 ± 190 µs | 13350 ± 420 µs | 14510 ± 370 µs | 15330 ± 370 µs |
| **2M** | 18400 ± 120 µs | 18520 ± 140 µs | 19300 ± 180 µs | 22330 ± 310 µs | 25460 ± 190 µs | 26050 ± 470 µs |
| **4M** | 32120 ± 110 µs | 32242 ± 88 µs | 33720 ± 310 µs | 39580 ± 500 µs | 45461 ± 41 µs | 47060 ± 270 µs |
| **8M** | 59750 ± 230 µs | 59780 ± 260 µs | 62910 ± 550 µs | 76140 ± 310 µs | 86.7 ± 1.6 ms | 91890 ± 130 µs |
| **16M** | 114480 ± 100 µs | 114230 ± 250 µs | 121960 ± 290 µs | 143680 ± 160 µs | 166360 ± 570 µs | 171.9 ± 1.2 ms |

Table 20: Intersection time: Indexing into large bitmap – no split (ILBN)

| Sizes of sets | | Binary search (BSS) | Binary search with look-up (BSLS) | Interpolation search (ISS) | Interpolation search with look-up (ISLS) | Generalised quadratic search (GQSS) | Local search (jobs created on CPU) (LSJCCPUS) | Local search (jobs created on GPU) (LSJCGPUS) |
|---|---|---|---|---|---|---|---|---|
| **126976** | **4096** | 678 ± 13 µs | 671 ± 28 µs | 654 ± 13 µs | 677 ± 12 µs | 688 ± 13 µs | 873 ± 17 µs | 929 ± 45 µs |
| **4096** | **126976** | 683 ± 22 µs | 705 ± 46 µs | 661 ± 19 µs | 678 ± 35 µs | 649 ± 27 µs | 890 ± 22 µs | 939 ± 25 µs |
| **122880** | **8192** | 674 ± 33 µs | 669 ± 23 µs | 655 ± 18 µs | 730 ± 200 µs | 699 ± 57 µs | 886 ± 52 µs | 920 ± 300 µs |
| **8192** | **122880** | 672 ± 34 µs | 671 ± 26 µs | 662 ± 18 µs | 663 ± 23 µs | 666 ± 27 µs | 913 ± 25 µs | 946 ± 34 µs |
| **114688** | **16384** | 688 ± 15 µs | 670 ± 18 µs | 673 ± 14 µs | 682 ± 9.9 µs | 694 ± 14 µs | 898 ± 28 µs | 898 ± 24 µs |
| **16384** | **114688** | 691 ± 32 µs | 656 ± 12 µs | 657 ± 12 µs | 651 ± 11 µs | 658 ± 26 µs | 945 ± 30 µs | 961 ± 22 µs |
| **98304** | **32768** | 764 ± 19 µs | 779 ± 13 µs | 756 ± 23 µs | 785.2 ± 8.5 µs | 772 ± 21 µs | 971 ± 34 µs | 994 ± 20 µs |
| **32768** | **98304** | 694 ± 21 µs | 704 ± 33 µs | 679 ± 20 µs | 690 ± 22 µs | 690 ± 21 µs | 973 ± 17 µs | 987 ± 18 µs |
| **65536** | **65536** | 745 ± 12 µs | 746 ± 12 µs | 735 ± 11 µs | 747 ± 11 µs | 749.8 ± 8.3 µs | 984 ± 40 µs | 993 ± 31 µs |
| **520192** | **4096** | 1580 ± 110 µs | 1564.2 ± 8.8 µs | 1525.3 ± 9.6 µs | 1579 ± 8.7 µs | 1621 ± 15 µs | 1809 ± 10 µs | 1968 ± 25 µs |
| **4096** | **520192** | 1484 ± 12 µs | 1482 ± 12 µs | 1476 ± 16 µs | 1477 ± 16 µs | 1481 ± 13 µs | 1895 ± 11 µs | 2111 ± 31 µs |
| **516096** | **8192** | 1581 ± 12 µs | 1596 ± 16 µs | 1534.6 ± 8.6 µs | 1595.7 ± 9.6 µs | 1641 ± 11 µs | 1791 ± 37 µs | 1983 ± 22 µs |
| **8192** | **516096** | 1489 ± 13 µs | 1487 ± 10 µs | 1480 ± 11 µs | 1481 ± 12 µs | 1489 ± 18 µs | 1913 ± 13 µs | 2102 ± 28 µs |
| **507904** | **16384** | 1590.4 ± 9.6 µs | 1601 ± 11 µs | 1535 ± 16 µs | 1597 ± 14 µs | 1640 ± 15 µs | 1822 ± 29 µs | 1981 ± 30 µs |
| **16384** | **507904** | 1501 ± 15 µs | 1495.7 ± 8.8 µs | 1489 ± 11 µs | 1486.4 ± 9.8 µs | 1504.4 ± 9.9 µs | 1913 ± 12 µs | 2112 ± 32 µs |
| **491520** | **32768** | 1672 ± 33 µs | 1694 ± 12 µs | 1613 ± 39 µs | 1678 ± 12 µs | 1733 ± 14 µs | 1922 ± 47 µs | 2041 ± 48 µs |
| **32768** | **491520** | 1528 ± 18 µs | 1526 ± 18 µs | 1510 ± 12 µs | 1515 ± 11 µs | 1528 ± 11 µs | 1926 ± 29 µs | 2091 ± 41 µs |
| **458752** | **65536** | 1699 ± 13 µs | 1706 ± 14 µs | 1631 ± 17 µs | 1693 ± 21 µs | 1744 ± 16 µs | 1973 ± 42 µs | 2096 ± 16 µs |
| **65536** | **458752** | 1502 ± 12 µs | 1505 ± 17 µs | 1482 ± 18 µs | 1488.5 ± 9 µs | 1500 ± 10 µs | 1915 ± 49 µs | 2051 ± 15 µs |
| **393216** | **131072** | 1675 ± 38 µs | 1706 ± 31 µs | 1615 ± 36 µs | 1673 ± 29 µs | 1715 ± 34 µs | 1940 ± 34 µs | 2122 ± 57 µs |
| **131072** | **393216** | 1670 ± 77 µs | 1655 ± 34 µs | 1577 ± 15 µs | 1619 ± 13 µs | 1684 ± 43 µs | 1989 ± 45 µs | 2117 ± 16 µs |
| **262144** | **262144** | 1781 ± 19 µs | 1805 ± 34 µs | 1733 ± 19 µs | 1753 ± 45 µs | 1767 ± 37 µs | 2056 ± 38 µs | 2249 ± 53 µs |
| **2093056** | **4096** | 4934 ± 31 µs | 4962 ± 37 µs | 4728 ± 31 µs | 4955 ± 29 µs | 5107 ± 37 µs | 4978 ± 48 µs | 5549 ± 46 µs |
| **4096** | **2093056** | 4289 ± 34 µs | 4288 ± 35 µs | 4276 ± 44 µs | 4263 ± 37 µs | 4271 ± 35 µs | 5101 ± 56 µs | 5821 ± 57 µs |
| **2088960** | **8192** | 4987 ± 31 µs | 5047 ± 30 µs | 4750 ± 18 µs | 5015 ± 33 µs | 5186 ± 38 µs | 5025 ± 44 µs | 5646 ± 30 µs |
| **8192** | **2088960** | 4300 ± 37 µs | 4314 ± 38 µs | 4269 ± 38 µs | 4272 ± 41 µs | 4296 ± 40 µs | 5183 ± 53 µs | 5860 ± 41 µs |
| **2080768** | **16384** | 5047 ± 35 µs | 5070 ± 34 µs | 4778 ± 31 µs | 4998 ± 23 µs | 5210 ± 38 µs | 5111 ± 72 µs | 5704 ± 40 µs |
| **16384** | **2080768** | 4329 ± 38 µs | 4328 ± 42 µs | 4289 ± 34 µs | 4301 ± 48 µs | 4325 ± 33 µs | 5216 ± 47 µs | 5894 ± 47 µs |
| **2064384** | **32768** | 5194 ± 33 µs | 5248 ± 34 µs | 4880 ± 29 µs | 5145 ± 37 µs | 5336 ± 41 µs | 5150 ± 100 µs | 5862 ± 63 µs |
| **32768** | **2064384** | 4589 ± 39 µs | 4572 ± 21 µs | 4563 ± 34 µs | 4579 ± 35 µs | 4588 ± 43 µs | 5299 ± 52 µs | 5965 ± 41 µs |
| **2031616** | **65536** | 5238 ± 42 µs | 5338 ± 40 µs | 4943 ± 37 µs | 5190 ± 34 µs | 5411 ± 34 µs | 5230 ± 120 µs | 5916 ± 65 µs |
| **65536** | **2031616** | 4607 ± 44 µs | 4594 ± 96 µs | 4527 ± 85 µs | 4548 ± 79 µs | 4621 ± 44 µs | 5277 ± 58 µs | 5923 ± 38 µs |
| **1966080** | **131072** | 5316 ± 42 µs | 5397 ± 49 µs | 4972 ± 43 µs | 5270 ± 40 µs | 5445 ± 38 µs | 5330 ± 120 µs | 6009 ± 69 µs |
| **131072** | **1966080** | 4795 ± 29 µs | 4840 ± 200 µs | 4790 ± 120 µs | 4830 ± 53 µs | 4862 ± 53 µs | 5515 ± 58 µs | 6143 ± 56 µs |
| **1835008** | **262144** | 5453 ± 73 µs | 5487 ± 34 µs | 5050 ± 61 µs | 5377 ± 35 µs | 5515 ± 38 µs | 5492 ± 91 µs | 6186 ± 65 µs |
| **262144** | **1835008** | 4822 ± 55 µs | 4813 ± 38 µs | 4715 ± 34 µs | 4775 ± 40 µs | 4802 ± 56 µs | 5447 ± 61 µs | 6130 ± 38 µs |
| **1572864** | **524288** | 5054 ± 40 µs | 5049 ± 25 µs | 4668 ± 23 µs | 4934 ± 46 µs | 5154 ± 39 µs | 5170 ± 110 µs | 5966 ± 49 µs |
| **524288** | **1572864** | 4697 ± 40 µs | 4752 ± 37 µs | 4566 ± 36 µs | 4697 ± 37 µs | 4710 ± 39 µs | 5175 ± 52 µs | 5880 ± 43 µs |
| **1048576** | **1048576** | 4778 ± 36 µs | 4835 ± 38 µs | 4532 ± 36 µs | 4760 ± 37 µs | 4816 ± 34 µs | 5090 ± 110 µs | 5900 ± 65 µs |

Table 21: Intersection time: Asymmetric sorted sets (part I)

| Sizes of sets | | Binary search (BSS) | Binary search with look-up (BSLS) | Interpolation search (ISS) | Interpolation search with look-up (ISLS) | Generalised quadratic search (GQSS) | Local search (jobs created on CPU) (LSJCCPUS) | Local search (jobs created on GPU) (LSJCGPUS) |
|---|---|---|---|---|---|---|---|---|
| **8384512** | **4096** | 19140 ± 60 µs | 19258 ± 76 µs | 18311 ± 53 µs | 19276 ± 71 µs | 19784 ± 62 µs | 17406 ± 87 µs | 19660 ± 140 µs |
| **4096** | **8384512** | 16286 ± 66 µs | 16256 ± 58 µs | 16260 ± 59 µs | 16258 ± 61 µs | 16220 ± 360 µs | 17448 ± 85 µs | 19960 ± 110 µs |
| **8380416** | **8192** | 19230 ± 180 µs | 19689 ± 54 µs | 18467 ± 58 µs | 19500 ± 190 µs | 20128 ± 78 µs | 17500 ± 110 µs | 19805 ± 93 µs |
| **8192** | **8380416** | 16248 ± 48 µs | 16284 ± 58 µs | 16353 ± 52 µs | 16570 ± 120 µs | 16612 ± 70 µs | 17710 ± 170 µs | 20480 ± 120 µs |
| **8372224** | **16384** | 19529 ± 66 µs | 20013 ± 58 µs | 18507 ± 64 µs | 19790 ± 120 µs | 20155 ± 60 µs | 17780 ± 110 µs | 20120 ± 110 µs |
| **16384** | **8372224** | 16350 ± 58 µs | 16323 ± 64 µs | 16246 ± 68 µs | 16260 ± 55 µs | 16323 ± 65 µs | 17920 ± 110 µs | 20580 ± 110 µs |
| **8355840** | **32768** | 19924 ± 73 µs | 20580 ± 190 µs | 18882 ± 99 µs | 20060 ± 260 µs | 20692 ± 64 µs | 18185 ± 66 µs | 20564 ± 67 µs |
| **32768** | **8355840** | 16720 ± 110 µs | 16690 ± 120 µs | 16600 ± 100 µs | 16590 ± 110 µs | 16680 ± 180 µs | 18596 ± 79 µs | 21180 ± 210 µs |
| **8323072** | **65536** | 20080 ± 390 µs | 20800 ± 180 µs | 18895 ± 72 µs | 20190 ± 100 µs | 20696 ± 66 µs | 18425 ± 68 µs | 20811 ± 72 µs |
| **65536** | **8323072** | 16752 ± 87 µs | 16790 ± 110 µs | 16660 ± 160 µs | 16670 ± 170 µs | 16780 ± 180 µs | 18788 ± 89 µs | 21350 ± 170 µs |
| **8257536** | **131072** | 20440 ± 74 µs | 20747 ± 95 µs | 18920 ± 200 µs | 20390 ± 190 µs | 20727 ± 71 µs | 18711 ± 70 µs | 21040 ± 220 µs |
| **131072** | **8257536** | 16985 ± 98 µs | 17000 ± 130 µs | 16870 ± 100 µs | 16810 ± 95 µs | 17072 ± 86 µs | 18893 ± 79 µs | 21344 ± 74 µs |
| **8126464** | **262144** | 20629 ± 57 µs | 21052 ± 64 µs | 19124 ± 62 µs | 20120 ± 69 µs | 21006 ± 68 µs | 19025 ± 56 µs | 21453 ± 76 µs |
| **262144** | **8126464** | 17630 ± 450 µs | 17440 ± 340 µs | 17440 ± 240 µs | 17600 ± 170 µs | 17981 ± 68 µs | 19521 ± 91 µs | 22100 ± 140 µs |
| **7864320** | **524288** | 20416 ± 51 µs | 20900 ± 65 µs | 18757 ± 60 µs | 20040 ± 200 µs | 20721 ± 70 µs | 18985 ± 59 µs | 21570 ± 180 µs |
| **524288** | **7864320** | 17228 ± 63 µs | 17300 ± 290 µs | 16964 ± 62 µs | 17170 ± 43 µs | 17310 ± 200 µs | 18993 ± 88 µs | 21570 ± 100 µs |
| **7340032** | **1048576** | 20290 ± 150 µs | 20932 ± 67 µs | 18709 ± 58 µs | 20143 ± 86 µs | 20530 ± 63 µs | 19004 ± 63 µs | 21499 ± 81 µs |
| **1048576** | **7340032** | 17118 ± 59 µs | 17071 ± 54 µs | 16701 ± 55 µs | 16920 ± 55 µs | 16962 ± 48 µs | 18725 ± 79 µs | 21650 ± 230 µs |
| **6291456** | **2097152** | 19901 ± 57 µs | 20282 ± 58 µs | 18418 ± 57 µs | 19872 ± 48 µs | 20033 ± 60 µs | 18926 ± 65 µs | 21783 ± 94 µs |
| **2097152** | **6291456** | 17498 ± 56 µs | 17707 ± 57 µs | 16920 ± 53 µs | 17580 ± 59 µs | 17457 ± 53 µs | 18325 ± 68 µs | 21124 ± 67 µs |
| **4194304** | **4194304** | 18663 ± 52 µs | 19387 ± 57 µs | 17604 ± 58 µs | 18944 ± 55 µs | 18670 ± 50 µs | 18610 ± 190 µs | 21480 ± 230 µs |
| **33550336** | **4096** | 75210 ± 170 µs | 75660 ± 230 µs | 72500 ± 190 µs | 76290 ± 190 µs | 78810 ± 230 µs | 66610 ± 200 µs | 75520 ± 470 µs |
| **4096** | **33550336** | 63160 ± 240 µs | 63240 ± 190 µs | 63280 ± 230 µs | 63190 ± 220 µs | 63230 ± 160 µs | 67270 ± 950 µs | 77670 ± 490 µs |
| **33546240** | **8192** | 75830 ± 220 µs | 78260 ± 230 µs | 72350 ± 170 µs | 77400 ± 190 µs | 78810 ± 160 µs | 66070 ± 240 µs | 75680 ± 480 µs |
| **8192** | **33546240** | 63320 ± 140 µs | 63250 ± 170 µs | 63250 ± 160 µs | 63230 ± 180 µs | 63260 ± 180 µs | 66320 ± 150 µs | 76730 ± 480 µs |
| **33538048** | **16384** | 76640 ± 150 µs | 78630 ± 180 µs | 72220 ± 180 µs | 77690 ± 200 µs | 78700 ± 190 µs | 66270 ± 410 µs | 76400 ± 740 µs |
| **16384** | **33538048** | 63330 ± 190 µs | 63400 ± 170 µs | 63300 ± 170 µs | 63310 ± 200 µs | 63260 ± 180 µs | 67080 ± 170 µs | 78.6 ± 1 ms |
| **33521664** | **32768** | 78400 ± 750 µs | 80300 ± 880 µs | 73390 ± 560 µs | 77920 ± 960 µs | 80500 ± 720 µs | 68080 ± 760 µs | 76580 ± 240 µs |
| **32768** | **33521664** | 64.5 ± 1.6 ms | 64210 ± 550 µs | 64060 ± 550 µs | 64050 ± 570 µs | 64150 ± 570 µs | 68370 ± 600 µs | 78.2 ± 1.4 ms |
| **33488896** | **65536** | 78990 ± 580 µs | 81390 ± 570 µs | 73490 ± 590 µs | 77640 ± 850 µs | 80570 ± 570 µs | 69150 ± 710 µs | 77850 ± 220 µs |
| **65536** | **33488896** | 64360 ± 570 µs | 64240 ± 540 µs | 64090 ± 560 µs | 64060 ± 560 µs | 64.8 ± 1 ms | 70060 ± 900 µs | 79480 ± 150 µs |
| **33423360** | **131072** | 79860 ± 830 µs | 81540 ± 850 µs | 73840 ± 560 µs | 78780 ± 850 µs | 81060 ± 580 µs | 70690 ± 740 µs | 79300 ± 190 µs |
| **131072** | **33423360** | 64500 ± 630 µs | 64490 ± 570 µs | 64180 ± 560 µs | 64020 ± 990 µs | 64420 ± 560 µs | 70440 ± 580 µs | 83530 ± 480 µs |
| **33292288** | **262144** | 80570 ± 280 µs | 82450 ± 200 µs | 73510 ± 200 µs | 80010 ± 780 µs | 81200 ± 510 µs | 72.6 ± 1.1 ms | 82 ± 1.1 ms |
| **262144** | **33292288** | 64710 ± 160 µs | 64730 ± 130 µs | 64590 ± 630 µs | 65 ± 1.1 ms | 65940 ± 120 µs | 72930 ± 660 µs | 82440 ± 900 µs |
| **33030144** | **524288** | 81020 ± 240 µs | 83020 ± 200 µs | 73620 ± 190 µs | 79660 ± 200 µs | 81370 ± 210 µs | 71620 ± 230 µs | 81620 ± 730 µs |
| **524288** | **33030144** | 65370 ± 160 µs | 65440 ± 180 µs | 64900 ± 180 µs | 64840 ± 180 µs | 65780 ± 170 µs | 72080 ± 210 µs | 82890 ± 780 µs |
| **32505856** | **1048576** | 81780 ± 210 µs | 83540 ± 250 µs | 73930 ± 200 µs | 80160 ± 240 µs | 81810 ± 190 µs | 72330 ± 200 µs | 82440 ± 760 µs |
| **1048576** | **32505856** | 65980 ± 130 µs | 65990 ± 140 µs | 65540 ± 160 µs | 65370 ± 210 µs | 66880 ± 240 µs | 72210 ± 210 µs | 83.5 ± 1.2 ms |
| **31457280** | **2097152** | 82030 ± 240 µs | 83700 ± 200 µs | 74130 ± 190 µs | 81080 ± 200 µs | 82180 ± 200 µs | 72980 ± 320 µs | 83130 ± 700 µs |
| **2097152** | **31457280** | 66290 ± 140 µs | 66640 ± 150 µs | 65740 ± 170 µs | 66350 ± 150 µs | 67160 ± 270 µs | 72410 ± 200 µs | 83480 ± 850 µs |
| **29360128** | **4194304** | 81620 ± 180 µs | 86260 ± 380 µs | 73780 ± 250 µs | 83920 ± 460 µs | 81210 ± 190 µs | 73580 ± 760 µs | 83980 ± 980 µs |
| **4194304** | **29360128** | 66910 ± 160 µs | 67140 ± 210 µs | 65230 ± 210 µs | 66840 ± 210 µs | 66320 ± 160 µs | 71340 ± 340 µs | 82830 ± 910 µs |
| **25165824** | **8388608** | 79570 ± 210 µs | 84230 ± 220 µs | 72240 ± 150 µs | 81960 ± 190 µs | 78760 ± 180 µs | 72820 ± 310 µs | 83870 ± 620 µs |
| **8388608** | **25165824** | 68920 ± 160 µs | 72800 ± 150 µs | 66230 ± 220 µs | 72200 ± 140 µs | 68360 ± 140 µs | 70750 ± 520 µs | 82580 ± 850 µs |
| **16777216** | **16777216** | 74170 ± 160 µs | 80500 ± 210 µs | 69050 ± 170 µs | 78520 ± 210 µs | 73270 ± 180 µs | 71120 ± 240 µs | 83030 ± 680 µs |

Table 22: Intersection time: Asymmetric sorted sets (part II)

| Sizes of sets | | Linear hashing with buffered output (LHB) | Linear hashing with sorted keys (LHS) | Cuckoo hashing to global table (CHG) | Cuckoo hashing to local table (CHL) | Indexing into big bitmap with buffered split (ILBS) | Indexing into big bitmap - no split (ILBN) |
|---|---|---|---|---|---|---|---|
| **126976** | **4096** | 979 ± 45 µs | 981 ± 44 µs | 1301 ± 11 µs | 1750 ± 360 µs | 5588 ± 43 µs | 4678 ± 16 µs |
| **4096** | **126976** | 872 ± 12 µs | 856 ± 19 µs | 1200 ± 12 µs | 1659 ± 26 µs | 5203 ± 19 µs | 4673 ± 13 µs |
| **122880** | **8192** | 938 ± 21 µs | 937 ± 16 µs | 1360 ± 190 µs | 1662 ± 13 µs | 5532 ± 24 µs | 4648 ± 14 µs |
| **8192** | **122880** | 835 ± 14 µs | 831 ± 30 µs | 1323 ± 16 µs | 1819 ± 30 µs | 5224 ± 17 µs | 4698 ± 13 µs |
| **114688** | **16384** | 956 ± 11 µs | 956.1 ± 8.3 µs | 1297 ± 18 µs | 1709 ± 39 µs | 5430 ± 14 µs | 4632 ± 18 µs |
| **16384** | **114688** | 892 ± 12 µs | 880 ± 12 µs | 1257 ± 19 µs | 1660 ± 19 µs | 5201 ± 12 µs | 4696 ± 11 µs |
| **98304** | **32768** | 987 ± 16 µs | 988.3 ± 9.1 µs | 1313 ± 12 µs | 1693 ± 15 µs | 5414 ± 15 µs | 4704 ± 12 µs |
| **32768** | **98304** | 956 ± 17 µs | 953 ± 22 µs | 1268 ± 12 µs | 1673 ± 15 µs | 5263 ± 15 µs | 4681.6 ± 9.8 µs |
| **65536** | **65536** | 946.5 ± 8.1 µs | 941 ± 11 µs | 1299.4 ± 8 µs | 1722 ± 15 µs | 5316 ± 12 µs | 4696 ± 10 µs |
| **520192** | **4096** | 2370 ± 110 µs | 2338 ± 12 µs | 2840 ± 120 µs | 3040 ± 21 µs | 7087 ± 17 µs | 5874 ± 19 µs |
| **4096** | **520192** | 1799 ± 21 µs | 1773 ± 18 µs | 2253 ± 26 µs | 2996 ± 37 µs | 6466 ± 27 µs | 6093 ± 26 µs |
| **516096** | **8192** | 2405 ± 82 µs | 2365 ± 24 µs | 2783 ± 6.1 µs | 2964 ± 10 µs | 7068 ± 21 µs | 5867 ± 14 µs |
| **8192** | **516096** | 1806 ± 11 µs | 1763.2 ± 8.5 µs | 2372 ± 11 µs | 2909 ± 15 µs | 6475 ± 19 µs | 6099 ± 14 µs |
| **507904** | **16384** | 2307 ± 14 µs | 2309 ± 11 µs | 2781.5 ± 7.4 µs | 2962 ± 13 µs | 6928 ± 15 µs | 5849.6 ± 7.4 µs |
| **16384** | **507904** | 1827.3 ± 9.5 µs | 1782.6 ± 7.5 µs | 2443 ± 11 µs | 2897 ± 14 µs | 6455 ± 21 µs | 6089 ± 13 µs |
| **491520** | **32768** | 2458 ± 14 µs | 2456 ± 15 µs | 2889 ± 12 µs | 3066 ± 19 µs | 6962 ± 10 µs | 6051 ± 11 µs |
| **32768** | **491520** | 1861.9 ± 9.4 µs | 1815.9 ± 7 µs | 2516 ± 10 µs | 2919 ± 13 µs | 6495 ± 22 µs | 6105 ± 14 µs |
| **458752** | **65536** | 2438 ± 18 µs | 2428 ± 16 µs | 2912 ± 11 µs | 3060 ± 23 µs | 6872 ± 12 µs | 6075 ± 15 µs |
| **65536** | **458752** | 1962 ± 17 µs | 1920 ± 13 µs | 2634 ± 12 µs | 2910 ± 18 µs | 6463 ± 22 µs | 6095 ± 49 µs |
| **393216** | **131072** | 2477 ± 45 µs | 2437 ± 26 µs | 2867 ± 19 µs | 3103 ± 43 µs | 7019 ± 31 µs | 6155 ± 31 µs |
| **131072** | **393216** | 2110 ± 14 µs | 2065.2 ± 8.4 µs | 2860 ± 140 µs | 3047 ± 55 µs | 6695 ± 20 µs | 6162 ± 54 µs |
| **262144** | **262144** | 2287 ± 33 µs | 2256 ± 37 µs | 3014 ± 14 µs | 3082 ± 14 µs | 6908 ± 15 µs | 6268 ± 32 µs |
| **2093056** | **4096** | 8270 ± 140 µs | 8141 ± 44 µs | 8853 ± 48 µs | 13721 ± 87 µs | 13950 ± 60 µs | 11460 ± 57 µs |
| **4096** | **2093056** | 5395 ± 91 µs | 5190 ± 43 µs | 6547 ± 56 µs | 13315 ± 86 µs | 12454 ± 61 µs | 11795 ± 53 µs |
| **2088960** | **8192** | 8200 ± 130 µs | 8126 ± 44 µs | 8862 ± 55 µs | 13773 ± 74 µs | 13904 ± 58 µs | 11461 ± 52 µs |
| **8192** | **2088960** | 5527 ± 40 µs | 5347 ± 33 µs | 7018 ± 43 µs | 13340 ± 81 µs | 12487 ± 42 µs | 11799 ± 50 µs |
| **2080768** | **16384** | 8102 ± 34 µs | 8097 ± 37 µs | 8872 ± 52 µs | 13838 ± 68 µs | 13752 ± 59 µs | 11434 ± 61 µs |
| **16384** | **2080768** | 5720 ± 110 µs | 5413 ± 35 µs | 7299 ± 47 µs | 13366 ± 84 µs | 12475 ± 59 µs | 11778 ± 47 µs |
| **2064384** | **32768** | 8316 ± 44 µs | 8314 ± 42 µs | 8970 ± 120 µs | 13977 ± 93 µs | 13722 ± 97 µs | 11519 ± 94 µs |
| **32768** | **2064384** | 5850 ± 35 µs | 5593 ± 20 µs | 7475 ± 35 µs | 13630 ± 100 µs | 12540 ± 160 µs | 11790 ± 140 µs |
| **2031616** | **65536** | 8350 ± 95 µs | 8301 ± 41 µs | 8951 ± 91 µs | 13980 ± 89 µs | 13636 ± 96 µs | 11520 ± 100 µs |
| **65536** | **2031616** | 5863 ± 59 µs | 5870 ± 110 µs | 7820 ± 110 µs | 13490 ± 94 µs | 12380 ± 83 µs | 11649 ± 19 µs |
| **1966080** | **131072** | 8293 ± 59 µs | 8263 ± 48 µs | 8990 ± 76 µs | 14190 ± 110 µs | 13716 ± 90 µs | 11590 ± 93 µs |
| **131072** | **1966080** | 6440 ± 230 µs | 5996 ± 27 µs | 8340 ± 150 µs | 13840 ± 150 µs | 12770 ± 110 µs | 11860 ± 130 µs |
| **1835008** | **262144** | 8235 ± 46 µs | 8189 ± 47 µs | 9248 ± 94 µs | 13770 ± 110 µs | 13681 ± 88 µs | 11820 ± 100 µs |
| **262144** | **1835008** | 6746 ± 53 µs | 6512 ± 55 µs | 9098 ± 91 µs | 14104 ± 80 µs | 12840 ± 130 µs | 12090 ± 180 µs |
| **1572864** | **524288** | 7752 ± 98 µs | 7602 ± 42 µs | 8798 ± 90 µs | 13740 ± 100 µs | 13194 ± 99 µs | 11550 ± 110 µs |
| **524288** | **1572864** | 6904 ± 50 µs | 6651 ± 32 µs | 9014 ± 56 µs | 13462 ± 82 µs | 12716 ± 53 µs | 11722 ± 49 µs |
| **1048576** | **1048576** | 7238 ± 46 µs | 7073 ± 44 µs | 9010 ± 100 µs | 7514 ± 90 µs | 12830 ± 99 µs | 11540 ± 110 µs |

Table 23: Intersection time: Asymmetric unsorted sets (part I)

| Sizes of sets | | Linear hashing with buffered output (LHB) | Linear hashing with sorted keys (LHS) | Cuckoo hashing to global table (CHG) | Cuckoo hashing to local table (CHL) | Indexing into big bitmap with buffered split (ILBS) | Indexing into big bitmap - no split (ILBN) |
|---|---|---|---|---|---|---|---|
| **8384512** | **4096** | 32040 ± 110 µs | 32030 ± 110 µs | 32960 ± 170 µs | 41920 ± 160 µs | 41010 ± 160 µs | 33490 ± 140 µs |
| **4096** | **8384512** | 20714 ± 78 µs | 19868 ± 67 µs | 23660 ± 260 µs | 39860 ± 280 µs | 36220 ± 250 µs | 34440 ± 280 µs |
| **8380416** | **8192** | 32500 ± 200 µs | 31980 ± 120 µs | 33030 ± 240 µs | 42080 ± 240 µs | 41070 ± 230 µs | 33820 ± 340 µs |
| **8192** | **8380416** | 21399 ± 83 µs | 20400 ± 72 µs | 25650 ± 240 µs | 39670 ± 260 µs | 36250 ± 250 µs | 34420 ± 250 µs |
| **8372224** | **16384** | 32060 ± 210 µs | 31980 ± 100 µs | 33170 ± 340 µs | 42540 ± 250 µs | 40950 ± 290 µs | 33490 ± 240 µs |
| **16384** | **8372224** | 21378 ± 50 µs | 20664 ± 60 µs | 26810 ± 260 µs | 40170 ± 250 µs | 36200 ± 220 µs | 34420 ± 260 µs |
| **8355840** | **32768** | 32270 ± 280 µs | 32570 ± 360 µs | 33130 ± 290 µs | 42200 ± 120 µs | 40760 ± 130 µs | 33590 ± 160 µs |
| **32768** | **8355840** | 21980 ± 220 µs | 21270 ± 210 µs | 27421 ± 96 µs | 40120 ± 110 µs | 36240 ± 100 µs | 34480 ± 110 µs |
| **8323072** | **65536** | 32670 ± 330 µs | 32300 ± 260 µs | 33000 ± 170 µs | 42590 ± 140 µs | 41010 ± 110 µs | 33550 ± 110 µs |
| **65536** | **8323072** | 22150 ± 210 µs | 21390 ± 190 µs | 28570 ± 110 µs | 40190 ± 120 µs | 36210 ± 120 µs | 34370 ± 100 µs |
| **8257536** | **131072** | 32300 ± 260 µs | 32250 ± 240 µs | 33470 ± 110 µs | 42350 ± 130 µs | 41190 ± 160 µs | 33640 ± 190 µs |
| **131072** | **8257536** | 23060 ± 250 µs | 22220 ± 220 µs | 30300 ± 150 µs | 40640 ± 140 µs | 36880 ± 150 µs | 34360 ± 170 µs |
| **8126464** | **262144** | 32500 ± 110 µs | 32460 ± 110 µs | 33490 ± 200 µs | 42470 ± 210 µs | 40910 ± 250 µs | 33940 ± 250 µs |
| **262144** | **8126464** | 24830 ± 110 µs | 24190 ± 130 µs | 33000 ± 270 µs | 41050 ± 310 µs | 37280 ± 370 µs | 35120 ± 590 µs |
| **7864320** | **524288** | 32480 ± 230 µs | 31880 ± 110 µs | 33430 ± 350 µs | 42340 ± 240 µs | 40420 ± 230 µs | 33620 ± 210 µs |
| **524288** | **7864320** | 25530 ± 130 µs | 23904 ± 81 µs | 34240 ± 270 µs | 40620 ± 230 µs | 36720 ± 230 µs | 34560 ± 230 µs |
| **7340032** | **1048576** | 31490 ± 110 µs | 30981 ± 95 µs | 33710 ± 210 µs | 42460 ± 210 µs | 40050 ± 200 µs | 33640 ± 210 µs |
| **1048576** | **7340032** | 25320 ± 100 µs | 24536 ± 75 µs | 34850 ± 250 µs | 40590 ± 220 µs | 36990 ± 250 µs | 34200 ± 220 µs |
| **6291456** | **2097152** | 30400 ± 330 µs | 30190 ± 120 µs | 33190 ± 200 µs | 42910 ± 210 µs | 39450 ± 190 µs | 33710 ± 230 µs |
| **2097152** | **6291456** | 27060 ± 230 µs | 25677 ± 74 µs | 34950 ± 210 µs | 41480 ± 180 µs | 37250 ± 190 µs | 34000 ± 170 µs |
| **4194304** | **4194304** | 28400 ± 100 µs | 27706 ± 92 µs | 34330 ± 250 µs | 44870 ± 270 µs | 38160 ± 240 µs | 33830 ± 240 µs |
| **33550336** | **4096** | 126440 ± 320 µs | 126350 ± 310 µs | 126600 ± 510 µs | 133520 ± 650 µs | 148770 ± 470 µs | 121030 ± 440 µs |
| **4096** | **33550336** | 79630 ± 210 µs | 77590 ± 190 µs | 91050 ± 500 µs | 122650 ± 450 µs | 129870 ± 420 µs | 123890 ± 440 µs |
| **33546240** | **8192** | 126840 ± 200 µs | 126820 ± 290 µs | 127.1 ± 2.6 ms | 133500 ± 480 µs | 148690 ± 330 µs | 120840 ± 290 µs |
| **8192** | **33546240** | 82900 ± 180 µs | 80180 ± 180 µs | 99040 ± 400 µs | 122450 ± 330 µs | 129800 ± 300 µs | 123780 ± 330 µs |
| **33538048** | **16384** | 126670 ± 350 µs | 126610 ± 240 µs | 126460 ± 340 µs | 133480 ± 460 µs | 148430 ± 250 µs | 120780 ± 230 µs |
| **16384** | **33538048** | 84190 ± 270 µs | 81270 ± 220 µs | 103400 ± 250 µs | 122640 ± 490 µs | 129660 ± 340 µs | 123740 ± 250 µs |
| **33521664** | **32768** | 126860 ± 420 µs | 126750 ± 270 µs | 126520 ± 300 µs | 133700 ± 540 µs | 148370 ± 200 µs | 120910 ± 220 µs |
| **32768** | **33521664** | 85700 ± 730 µs | 81970 ± 240 µs | 105700 ± 200 µs | 129.3 ± 1.1 ms | 130490 ± 770 µs | 124250 ± 650 µs |
| **33488896** | **65536** | 126830 ± 230 µs | 126760 ± 250 µs | 127.7 ± 1.5 ms | 134060 ± 270 µs | 148600 ± 400 µs | 121170 ± 200 µs |
| **65536** | **33488896** | 87560 ± 310 µs | 83090 ± 510 µs | 111580 ± 570 µs | 123.7 ± 1 ms | 131110 ± 650 µs | 123550 ± 170 µs |
| **33423360** | **131072** | 126850 ± 320 µs | 126760 ± 260 µs | 126890 ± 490 µs | 134290 ± 480 µs | 148580 ± 230 µs | 121040 ± 410 µs |
| **131072** | **33423360** | 88620 ± 540 µs | 85000 ± 260 µs | 115340 ± 300 µs | 126830 ± 510 µs | 129880 ± 240 µs | 123750 ± 240 µs |
| **33292288** | **262144** | 128800 ± 340 µs | 127950 ± 840 µs | 127.2 ± 1.5 ms | 134330 ± 920 µs | 148420 ± 250 µs | 121980 ± 730 µs |
| **262144** | **33292288** | 93710 ± 190 µs | 90520 ± 150 µs | 124980 ± 350 µs | 127690 ± 640 µs | 130270 ± 520 µs | 124610 ± 750 µs |
| **33030144** | **524288** | 126430 ± 260 µs | 126320 ± 260 µs | 127 ± 2.4 ms | 132120 ± 290 µs | 147880 ± 250 µs | 120850 ± 240 µs |
| **524288** | **33030144** | 97160 ± 270 µs | 92140 ± 180 µs | 132530 ± 280 µs | 125260 ± 260 µs | 130070 ± 210 µs | 123680 ± 230 µs |
| **32505856** | **1048576** | 126140 ± 240 µs | 125890 ± 250 µs | 127020 ± 620 µs | 132310 ± 580 µs | 147690 ± 230 µs | 120930 ± 250 µs |
| **1048576** | **32505856** | 98980 ± 220 µs | 93870 ± 240 µs | 137100 ± 300 µs | 123190 ± 220 µs | 130030 ± 290 µs | 123560 ± 200 µs |
| **31457280** | **2097152** | 125180 ± 250 µs | 124750 ± 230 µs | 127580 ± 710 µs | 132020 ± 360 µs | 147030 ± 230 µs | 120960 ± 230 µs |
| **2097152** | **31457280** | 100760 ± 200 µs | 95350 ± 230 µs | 138590 ± 250 µs | 123720 ± 370 µs | 130740 ± 230 µs | 123380 ± 230 µs |
| **29360128** | **4194304** | 123950 ± 180 µs | 123090 ± 220 µs | 128610 ± 340 µs | 132350 ± 240 µs | 145740 ± 250 µs | 121120 ± 170 µs |
| **4194304** | **29360128** | 102270 ± 230 µs | 97570 ± 210 µs | 138330 ± 190 µs | 127600 ± 230 µs | 131510 ± 210 µs | 122860 ± 190 µs |
| **25165824** | **8388608** | 120530 ± 240 µs | 118920 ± 190 µs | 130140 ± 340 µs | 132280 ± 220 µs | 143380 ± 220 µs | 121270 ± 280 µs |
| **8388608** | **25165824** | 105940 ± 250 µs | 101780 ± 380 µs | 137380 ± 270 µs | 126600 ± 210 µs | 133950 ± 310 µs | 122550 ± 240 µs |
| **16777216** | **16777216** | 112360 ± 320 µs | 109610 ± 300 µs | 132840 ± 230 µs | 135490 ± 310 µs | 138160 ± 200 µs | 121470 ± 210 µs |

Table 24: Intersection time: Asymmetric unsorted sets (part II)

| Size | Binary search | Generalized quadratic search | Local search (jobs created on CPU) | Local search (jobs created on GPU) |
|---|---|---|---|---|
| **32M** | 147.8 ± 4.1 ms | 166.5 ± 7.2 ms | 148 ± 2.2 ms | 169.3 ± 3.2 ms |
| **64M** | 293.7 ± 6.2 ms | 331 ± 11 ms | 289.7 ± 3.2 ms | 333.8 ± 4.6 ms |
| **128M** | 587.3 ± 7.9 ms | 671 ± 13 ms | 576.5 ± 7.1 ms | 661.1 ± 8.8 ms |
| **256M** | 1180 ± 15 ms | 1358 ± 19 ms | 1143 ± 15 ms | 1327 ± 12 ms |
| **512M** | 2368 ± 22 ms | 2724 ± 21 ms | 2293 ± 21 ms | 2665 ± 33 ms |

Table 25: Intersection time: Multirun sorted template (MST) with various inner strategies

| Size | MBT + LHS | MLBS | MLBB |
|---|---|---|---|
| **32M** | 1561 ± 12 ms | 2026.5 ± 3 ms | 324.6 ± 7.3 ms |
| **64M** | 3129 ± 24 ms | 4017.9 ± 5.8 ms | 606.7 ± 9.4 ms |
| **128M** | 6281 ± 37 ms | 8026.9 ± 8.2 ms | 1128 ± 13 ms |
| **256M** | 12708 ± 84 ms | 16001 ± 18 ms | 2301 ± 38 ms |
| **512M** | 25670 ± 220 ms | 32420 ± 230 ms | 4400 ± 40 ms |

Table 26: Intersection time: Multirun hash-based strategies

# Appendix B: Enclosed DVD contents

- */bin* Executable binaries of the test programs
- */results*
  - *raw/* Raw output of the testing
  - *charts/* All charts both included and not included in this text
  - *r/* Source code in R language for generating the charts
- */src/* All source code (both C++ and OpenCL)
  - *Intersection/* Algorithms for intersection
  - *Sorting/* Algorithms for sorting
  - *Shared/* Code shared by all algorithms (testing framework)
- */test_data/* Data used for tests
- */thesis.pdf* This document in electronic version