KioxiaSort: Sorting 1TB by 89K Joules

Shintaro Sano, Kioxia Corporation, shintarou.sano@kioxia.com
Zaid Mahmoud, Princess Sumaya University for Technology, z.tobasy@gmail.com
Tomoya Suzuki, Kioxia Corporation, tomoya.suzuki@kioxia.com

Abstract

KioxiaSort is a fast and low-power sorting optimized under the requirements of 2019 Joule sort (10^{10} records, Indy) benchmark. KioxiaSort can sort 1TB of data in 9 min using 89K Joules – which means 112K records sorted/joule.

1. Introduction

This paper shows algorithm, implementation and evaluation results of our sorting named KioxiaSort. KioxiaSort deals with Joule sort benchmark (10^{10} records, Indy). **Figure 1** shows the previous scores of Joule Sort winners. The latest winner of Joule Sort was NTOSort [1], which sorts 1TB of data by consuming 168K Joules of energy using 16 Serial-ATA SSDs. With the help of the recent NVMe™ SSDs, our KioxiaSort consumes only 89K Joules to sort 1TB. Thus, in comparison with the latest winners, KioxiaSort achieves better score.

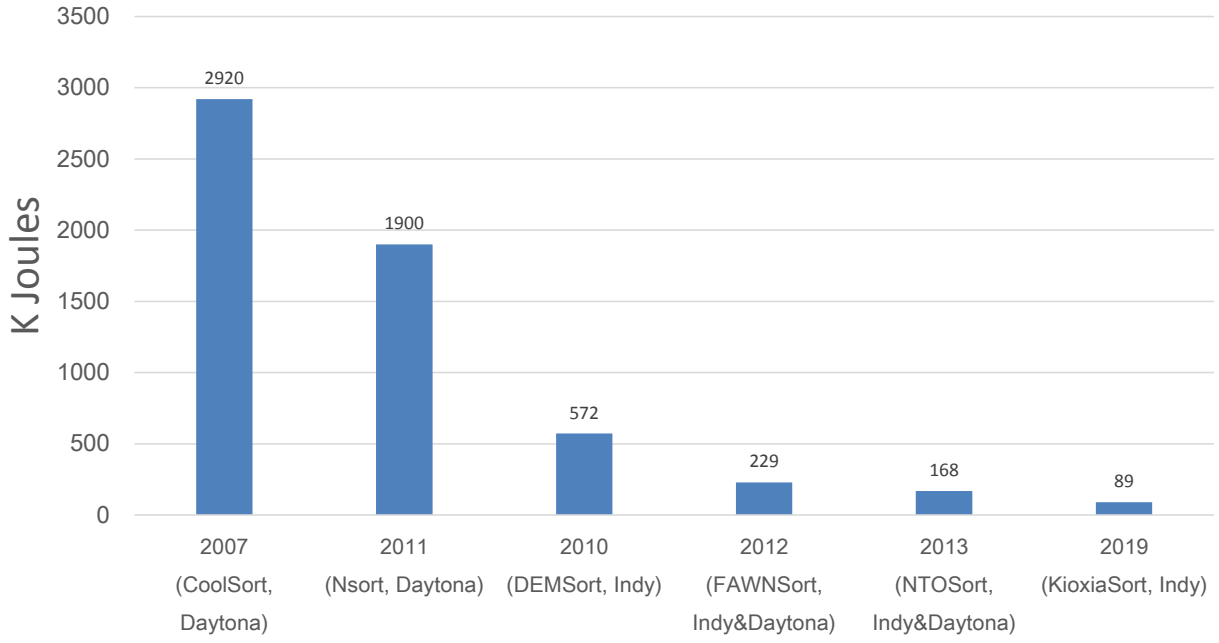

Figure 1: Yearly trends (2007- 2019) of the Joule Sort Benchmark (10^{10} records)

The remaining of this document are sections to give more details about how we designed, tested and evaluated this project. In Section 2, we describe our algorithm. Section 3 presents the hardware components and configurations. Section 4 describes the software components (OS, compiler and so on). Section 5 describes the evaluation procedure, and Section 6 shows the evaluation results.

2. Algorithm

KioxiaSort is based on the classical external merge sort. Please refer to the basics of external merge sort in "The Art of Computer Programming" [2] and Wikipedia [3], [4].

KioxiaSort algorithm consists of two main phases:

- **Run*** phase (Chunked sort)
  - Input data is read from disks and divided into "chunks". The size of each chunk is determined according to available RAM. Each chunk is sorted separately, and then written back into disks. More details about Run phase are shown in **Section 2.1.**
- **Merge** phase
  - The chunks (from Run phase) are read from disks, merged into final sorted data, and then written back into disks. More details about Merge phase are shown in **Section 2.2.**

2.1. Run Phase

Figure 2: Sorting pipeline

In the Run Phase, chunks are sorted in multi-threaded fashion (Each chunk is sorted by the corresponding thread). In order to utilize CPU resources maximally, thread executions are handled in a pipelined manner using mutual exclusions. Figure 2 shows an example of thread-pipeline for 4-core 12-thread execution (We use 8-core 24-thread in our final evaluation). In the figure, R denotes reading chunks from SSDs, S denotes sorting chunks, while W denotes writing sorted chunks into SSDs. Reading- and writing-steps do not consume

* In this context, *run* means *chunk,* not *execute* [2].

much CPU resources since they utilize DMA (with O_DIRECT†), and reading, writing and sorting are executed in parallel. In the sorting step, we also apply the techniques used in the past contests as below.

- DEMSort [5]: Indices are used instead of actual values in the sorting step. Also, radix sort is done for the first 2-byte of keys (The remaining bytes of keys are sorted by quicksort).
- OZsort [6]: The key data are arranged in little-endian for fast key comparison.

In our evaluation, chunk size is 800MiB. Dividing 1TB data set by 800MiB, the last chunk has valid entries less than 800MiB. In the last chunk, the valid entries are sorted, and the remaining space up to 800MiB is filled by dummy entries for the merge phase. As a result of Run phase, 1193 chunk files (each size 800MiB) are created.

2.2. Merge Phase

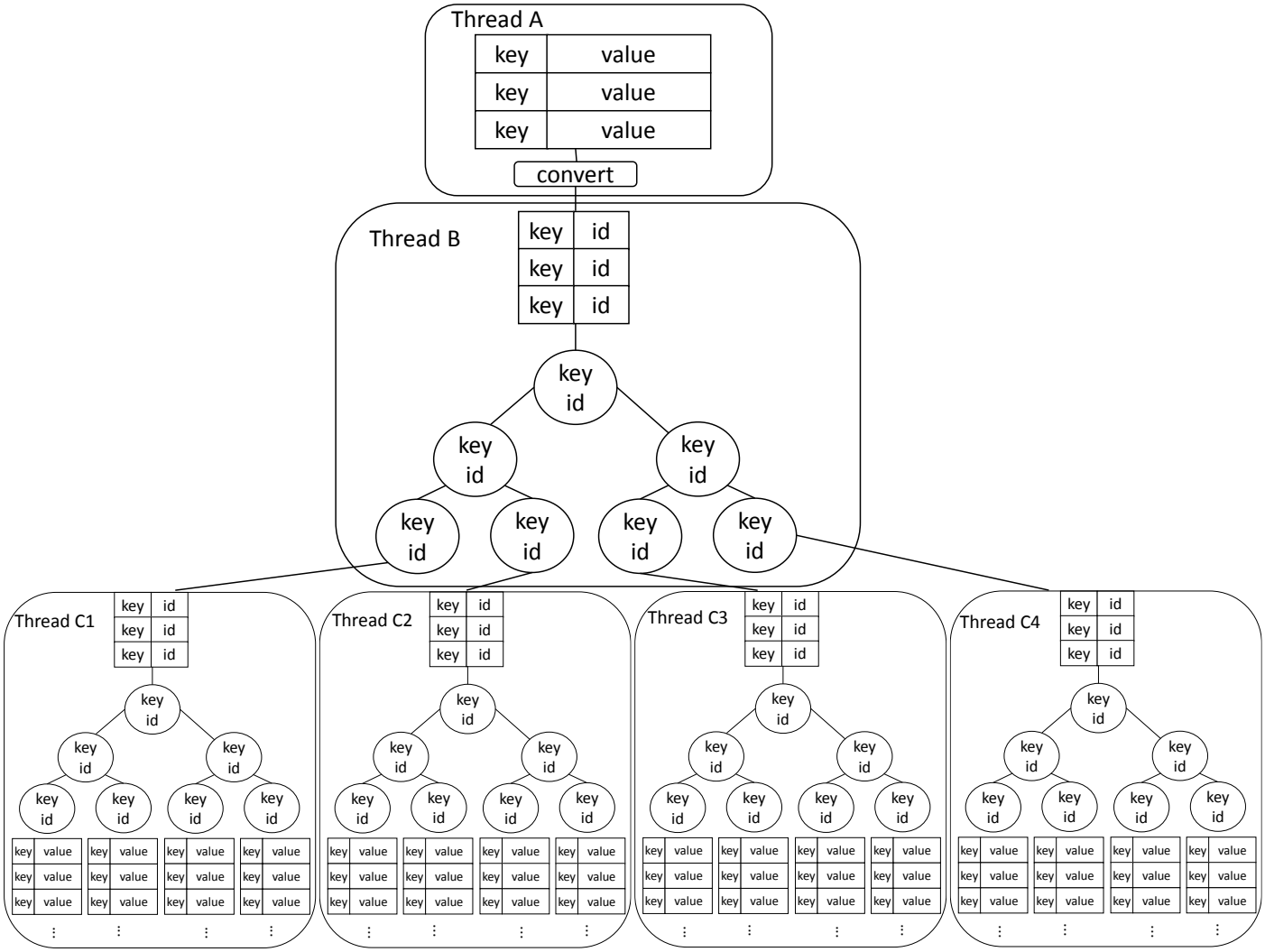

Figure 3: 2-stage Tournament Tree

In the Merge Phase, we use merge sort based on Tournament Tree [2], [4]. **Figure 3** demonstrates the merge

† For interested readers, the number of IO system calls is explained in Appendix A.

phase. It shows an example of merging 16 sorted chunks. In this figure, the input at the bottom represents the sorted chunks, while the output at the top represents the minimum key and the corresponding value. 2-stage merging is adopted to utilize multi-core CPU resources. Chunk IDs are used internally instead of actual values. The chunk IDs are finally converted into the original values. The conversion can be easily implemented since output values are accessed sequentially within each chunk. The execution of the merge phase is stopped when the valid 1TB data set is output. The dummy entries are not output since their keys are treated as bigger than any valid keys.

In our final evaluation, we use 6 threads for Thread C. Thus, thread B perform 6-way merge, while 1193 chunks are distributed on 6 threads as follows.

- Thread B : 6-way merge
- Thread C1-C5 : 199-way merge
- Thread C6 : 198-way merge

3. Hardware

Category		#	note
Motherboard	ASUS Prime Z370-A	1	
CPU	Intel Core i9-9900K	1	
Memory	Crucial 16GB DDR4-2666(CT16G4DFD8266)	4	Total: 64GB
Storage	Toshiba XG5-P KXG50PNV2T04 (2TB)	1	OS installed
	CFD CSSD-M2B1TPG3VNF (1TB)	8	
	Highpoint SSD7101A-1	1	
	ASUS Hyper M.2 x16	1	
Power source	KUROUTOSHIKOU KRPW-TI500W/94+	1	
Case Fan	Fractal Design Silent Series 120mm	2	
CPU cooler	SCYTHE Kotetsu Mark II	1	
Power meter	Hioki PW3335	1	

Table 1: Hardware List

We set up hardware components listed in **Table 1.** All of them are commercially available in the public market. In order to measure power consumption, Power meter Hioki PW3335 [7] is used. The power meter is compatible with the SPECpower® benchmark. The accuracy of PW3335 is within ±0.1% and it can synchronize through LAN, thus it satisfies the requirements for Joule Sort.

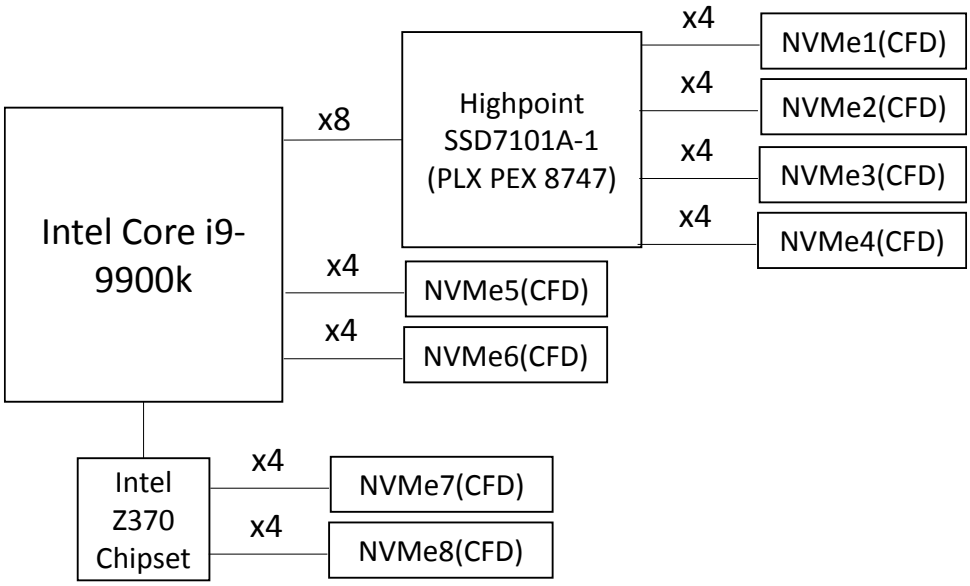

Figure 4: Block diagram of storage components

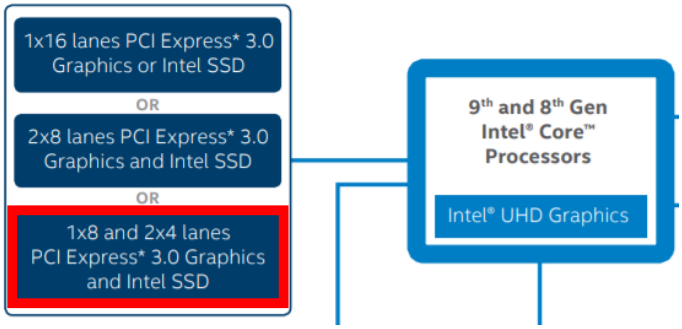

Figure 5: Intel Z370 Chipset Block Diagram [8]

Figure 4 shows block diagram of storage components. 8 NVMe™ SSDs are used. Intel Z370 chipset is configurable for PCIe lanes in three ways as shown in **Figure 5**. We choose 1x8 and 2x4 lanes outlined by the red box. This configuration is selected via BIOS as shown in Appendix D.

4. Software

OS	Ubuntu 19.04 Server	Config: Appendix C
Kernel	Linux 5.0.0-25-generic	
Compiler	gcc 9.1	-Ofast -march=native
RAID	LVM 2.02.176	Chunk size 512KB Config: Appendix B
File system	Ext4	

Table 2: Software List

Table 2 shows the software components for our evaluation. Ubuntu 19.04 Server and some necessary packages are installed. As of August 2019, the latest versions of the packages are used. OS configurations

are in default, except for the number of openable files, which is increased as explained in Appendix C.

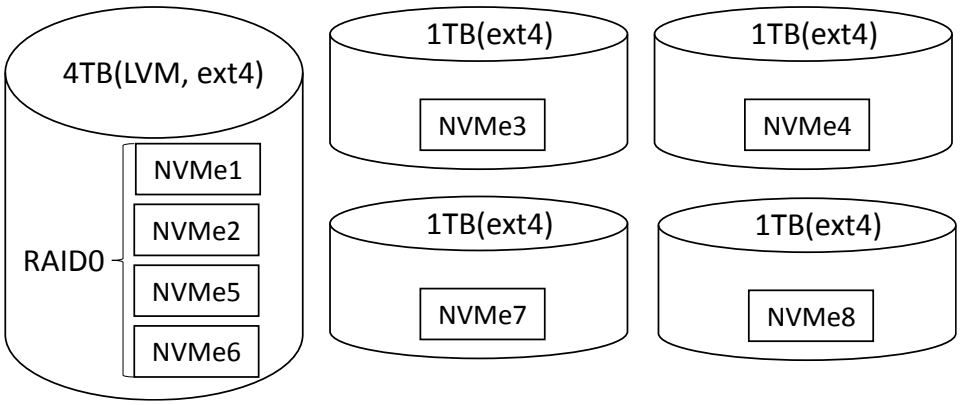

Figure 6: Volume configurations

Figure 6 shows volume configurations for storing sorting data (refer to section 7.2 for the reasons for this configuration). 5 volumes are prepared. One of them (4TB volume) uses LVM and RAID0 as explained in Appendix B.

5. Evaluation procedure

Joule Sort (10^{10} records, Indy) benchmark is evaluated as follows.

A. Prepare 1TB data set by using *gensort*.
B. Start power meter for logging.
C. Execute KioxiaSort.
- Execution time is measured during states from **Figure 7** to **Figure 9.**
- Input file is removed at the end of Run phase as shown in **Figure 8** (refer to section 7.3 for the reason). The removal is permitted in Indy sort.

D. Validate sorted data by using *valsort*.

Steps from A to D are repeated 5 times.

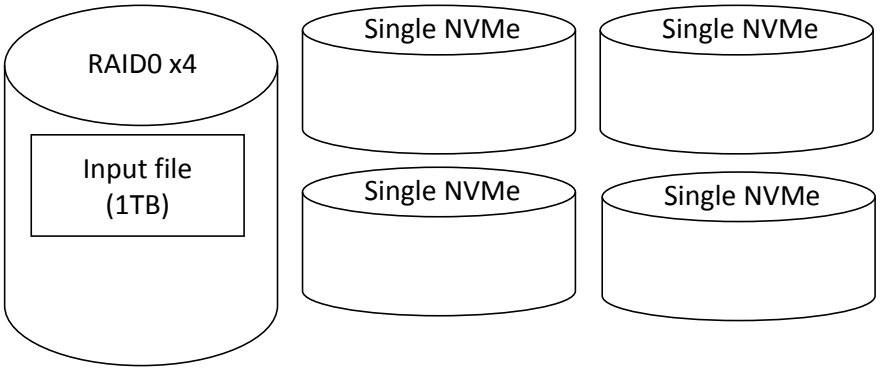

Figure 7: Initial State

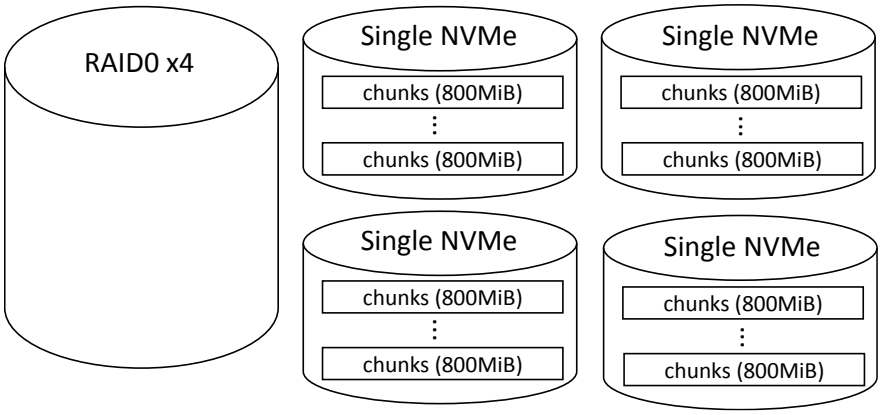

Figure 8: Run phase end state

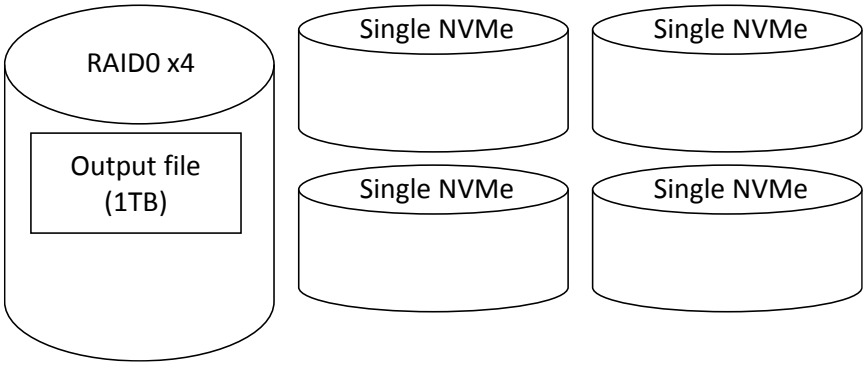

Figure 9: Final State

The power is measured as shown in **Figure 10**. The power supply cable of the sorting machine is plugged into the power meter. The consumed power of the sorting machine is measured by the power meter and logged to the monitoring PC via LAN at 0.2 second interval. The internal clock is synchronized between the sorting machine and the monitoring PC by NTP. The consumed power from start time to end time of KioxiaSort execution is reported as results in Section 6.

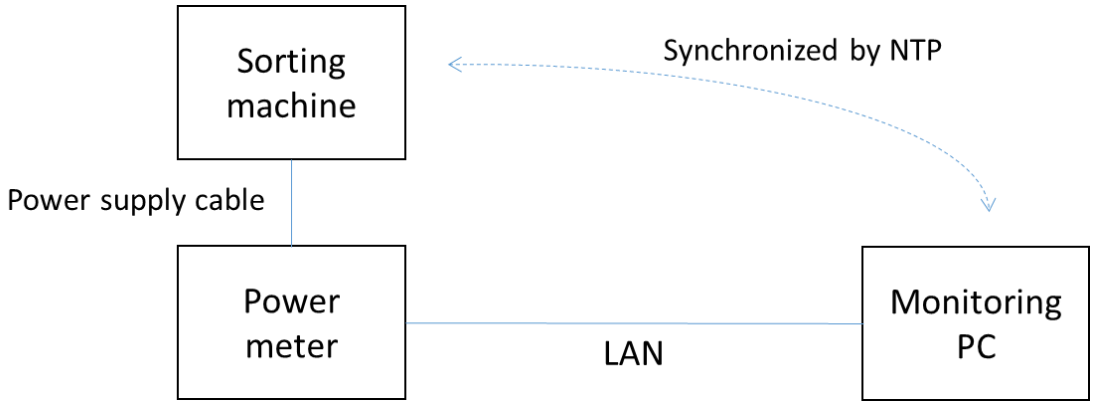

Figure 10: Power measurement environment

6. Results

The execution results of 5 times are shown in the following tables.

	Time(s)	Power(W)	Energy(J)	Srec/J	Valsort	Dup keys
Run 1	530.92	167.91	89,147	112,175	CRC: 12a06cd06eeb64b16	0
Run 2	523.67	169.52	88,774	112,645	CRC: 12a06cd06eeb64b16	0
Run 3	522.06	169.13	88,294	113,258	CRC: 12a06cd06eeb64b16	0
Run 4	523.52	170.01	89,004	112,355	CRC: 12a06cd06eeb64b16	0
Run 5	523.92	168.96	88,521	112,967	CRC: 12a06cd06eeb64b16	0
Avg	524.82	169.11	88,748	112,680		
stdev	3.48	0.782	347	395		

Table 3: Execution results

Table 4 shows the performance of the individual phases. "Time" and "CPU utilization" was measured by /bin/time command. Please note that "Avg. Power" is average of 180 sec for each phase (**Figure 11**), whereas "Power" in Table 3 is average of entire execution. Thus, "Avg. Power" and "Energy" in Table 4 are not the same as Table 3.

	Avg. Power (W)		Time (sec)		Energy (KJ)		CPU utilization (%)	
	Run	Merge	Run	Merge	Run	Merge	Run	Merge
Run1	173.5	169.3	277.5	240.1	48.1	40.6	848%	575%
Run2	173.6	169.5	278.1	240.1	48.3	40.7	863%	571%
Run3	173.7	166.4	277.2	240.1	48.1	39.9	857%	574%
Run4	173.7	165.7	278.9	240.1	48.4	39.8	856%	575%
Run5	171.4	169.1	277.7	241.5	47.6	40.8	854%	575%
Avg.	173.2	168.0	277.9	240.4	48.1	40.4	856%	574%

Table 4: Performance breakdown of individual phases

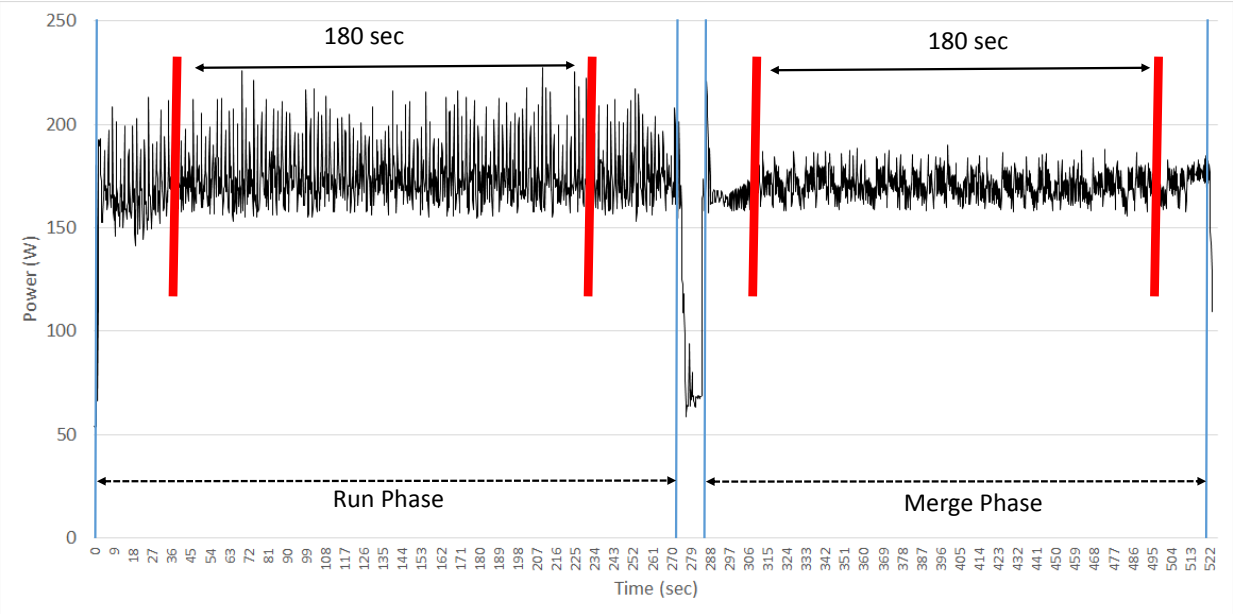

Figure 11: Power vs time (actual measurement result of Run 4)

7. Discussion

7.1. Single-thread merge vs. Multi-thread merge

As explained in Section 2.2, multi-threaded merge is employed. We have not done apple-to-apple comparison between single- and multi-threaded merge implementation. However, we estimated that the Joule score of multi-threaded implementation is 34% or more better than single-threaded one.

Table 5 shows our estimation. We've measured a single-threaded implementation (A) that uses blocking I/O, and multi-threaded implementation (C). We assume time and power of a single-threaded implementation (B) that uses non-blocking I/O as follows:

- the execution time of (B) is the same as user-time of (A)
- the power of (B) is more than that of (A)

As a consequence, we estimated that multi-threaded implementation (C) is 34% more efficient.

In our case, storage IO is not the bottleneck, thanks to 4 NVMe SSDs (i.e. more than 5.0 GB/s in total). As for IO among the threads, we've reduced the number of synchronization down to once per 1M-entry in order to improve performance.

	Time	Power	Joule	note
(A) Single thread merge (measured)	1146 sec	80.6 W	92.4 KJ	Blocking I/O
(B) Single thread merge (estimated)	665 sec	> 80.6 W	> 53.6 KJ	Non-blocking I/O
(C) 8 threads merge (measured)	240 sec	168.0 W	40.0 KJ	

Table 5: Performance comparison between single- and multi-thread merge

7.2. NVMe configuration

In Ssection 4, NVMe1-2-5-6 is selected for the RAID volume. This is because of the following reasons:

In case of the run phase, if we chose NVMe1-2-3-4 (**Figure 12**), the red arrow could be the bottleneck. The maximum bandwidth of RAID volume is limited by PCIe x8, and the peak performance of NVMe 1-2-3-4 could be limited.

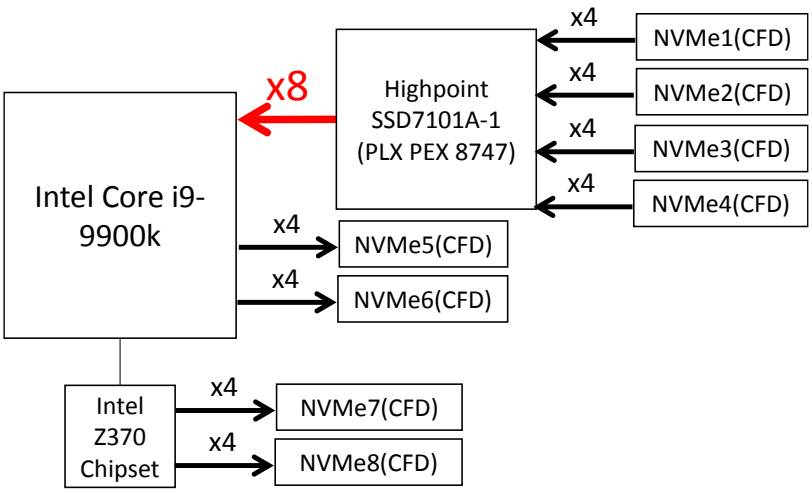

Figure 12: Data flow with RAID volume NVMe1-2-3-4

Figure 13 shows data flow of our configuration, in which the data is read from NVMe1-2-5-6 and written to NVMe3-4-7-8. In this case, the read performance from NVMe1-2-5-6 is not limited. Thanks to PCIe full-duplex transmission, the write performance to NVMe 3-4 is not limited as well.

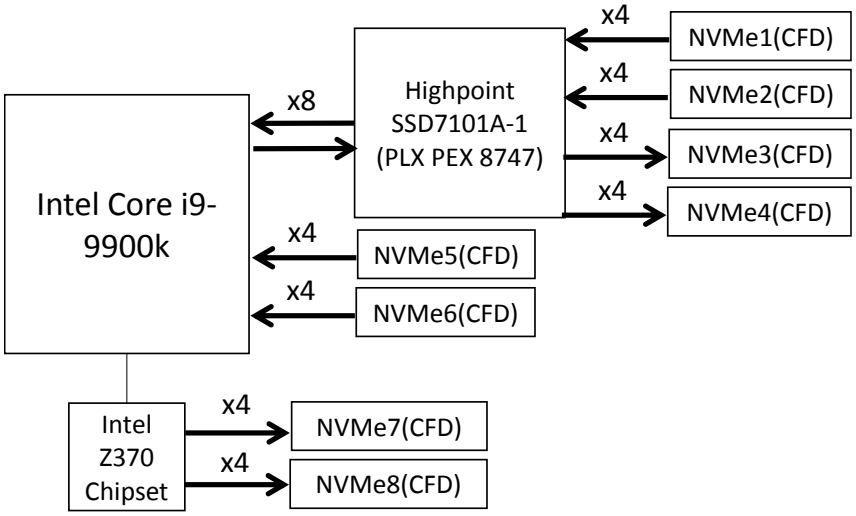

Figure 13: Data flow with RAID volume NVMe 1-2-5-6

7.3. Removal of the input file

In section 5, the input file is removed at the end of the Run phase. This is because of the following reasons:

In order to achieve better score in Joule sort, it is important for us to use up the performance of SSDs. In general, cleaning SSD's internal state is often beneficial for the performance. Thus, we deleted all files in the SSD. Also we used "fstrim" command to make sure the internal state be cleared. We performed some experiments both with/without the removal‡, and the performance with the removal was better. Please note that the time of the removal is included in the total execution time.

‡ We've compared "fio" write with "rm" and without "rm" (i.e. overwrite).

Acknowledgements

The authors would like to thank all the people related to the internship program and the collaboration with Princess Sumaya University for Technology for generous supports and helpful discussions.

We are grateful to Kazuhiro Hiwada-san who offered continuing support and constant encouragement.

We would like to thank the Sort Benchmark committee members for their insightful comments.

References

[1] A. Ebert, “NTOSort,” sortbenchmark.org, 2013.

[2] D. E. Knuth, The Art of Computer Programming: Volume 3: Sorting and Searching.

[3] "Wikipedia: External sorting," [Online]. Available: https://en.wikipedia.org/wiki/External_sorting.

[4] "Wikipedia: k-way merge algorithm," [Online]. Available: https://en.wikipedia.org/wiki/K-way_merge_algorithm.

[5] M. Rahn, "DEMSort — Distributed External Memory Sort," 2009.

[6] R. Sinha, "OzSort: Sorting 100GB for less than 87kJoules," 2009.

[7] "POWER METER PW335," Hioki, [Online]. Available: https://www.hioki.com/en/products/detail/?product_key=5598.

[8] Intel, "Intel (r) Z370 Chipset Product Brief," [Online]. Available: https://www.intel.co.jp/content/dam/www/public/us/en/documents/product-briefs/z370-chipset-product-brief.pdf.

Trademarks

Appendix A: The number of IO system calls

To mitigate system call overheads, we use relatively large IO size as follows.

In the Run Phase, both read and write system call carry 800MiB data except for the last read. For 1TB input file, reading by 800MiB is performed 1192 times (999,922,073,600 B in total), and the remaining 76,100 KiB (77,926,400B) is also read. 1193 temporary chunk files are created. The size of each chunk file is 800MiB. The total number of bytes for temporary files is 1,000,760,934,400 B.

In the Merge Phase, read system call is performed up to 152704 times[§] by 6.25 MiB (6,553,600 B), while write system call is performed 19073 times by 50 MiB (52,428,800 B), and one more write for the remaining 25,497,600B. The size of the output file is 1TB (1,000,000,000,000 B).

			IO size	#num	total
Run Phase	Read	Aligned	800 MiB	1192	999,922,073,600 B
		Unaligned	76,100 KiB	1	77,926,400 B
	Write		800 MiB	1193	1,000,760,934,400 B
Merge Phase	Read		6.25 MiB	Max 152704	Max 1,000,760,934,400 B
	Write	Aligned	50 MiB	19703	999,974,502,400 B
		Unaligned	24,900 KiB	1	25,497,600 B

Table 6: Summary of IOs.

Appendix B: RAID configurations

```
# pvcreate /dev/nvmeAn1 /dev/nvmeBn1 /dev/nvmeCn1 /dev/nvmeDn1
# vgcreate vg0 /dev/nvmeAn1 /dev/nvmeBn1 /dev/nvmeCn1 /dev/nvmeDn1
# lvcreate vg0 -l 95%VG -i4 -I 512K
# mkfs.ext4 /dev/mapper/vg0-lvol0
```

Appendix C: ulimit configuration

```
$ cat /etc/security/limits.conf
*               soft    nofile          65536
*               hard    nofile          65536
$ ulimit -n
65536
```

[§] The actual number of read is less than 152704 because the merge phase stops before all of dummy entries are read.

Appendix D: BIOS configuration

Hyper M.2X16 Enabled