Locality-Aware Stencil Computations using Flash SSDs as Main Memory Extension

Hiroko Midorikawa, Hideyuki Tan
Department of Computer and Information Science
Seikei University and JST CREST
Tokyo, Japan
midori@st.seikei.ac.jp

Abstract—This paper investigates the performance of flash solid state drives (SSDs) as an extension to main memory with a locality-aware algorithm for stencil computations. We propose three different configurations, swap, mmap, and aio, for accessing the flash media, with data structure blocking techniques. Our results indicate that hierarchical blocking optimizations for three tiers, flash SSD, DRAM, and cache, perform satisfactorily to bridge the DRAM-flash latency divide. Using only 32 GiB of DRAM and a flash SSD, with 7-point stencil computations for a 512 GiB problem (16 times that of the DRAM), 87% of the Mflops execution performance achieved with DRAM only was attained.

Keywords-Non-volatile memory; flash memory; memory hierarchy; tiling; temporal blocking; stencil; out-of-core; asynchronous IO; mmap; access locality; NUMA; auto-tuning;

I. INTRODUCTION

Scientific computation often requires significant amounts of memory to tackle large-scale problems and/or for higher resolution data analysis. One of the common solutions used to satisfy this requirement is aggregation of distributed memories over cluster nodes. This is typically accomplished by increasing the amount of DRAM per node and the number of nodes in a cluster. However, there is a limit on the extent to which DRAM can be increased in main memory, because the number of memory slots that can be accommodated on server boards is limited. Further, power consumption constraints and other resource limitations exist.

The advent of various kinds of Non-Volatile Memory (NVM) ushered in a new era in memory organization and memory-related software [1][2]. This new era influences not only the traditional memory hierarchy but also the basic idea of memory read/write and file IO. In addition, it has the potential to significantly change traditional programming models and application programs. Of various NVMs, flash memory is already widely available to end-users. Its access time is not as short as that of DRAM, but it provides much greater capacity at a lower cost and consumes less power. These points make flash memory one of the promising candidates for a DRAM extension to the main memory [3][4].

Recent PCIe bus-connected flash SSDs [5][6] achieve several hundred times faster latency than HDDs, but are still approximately one thousand times slower than DRAM. The gap between DRAM and flash SSD is much greater than the gap between L3 cache and memory, which has only a three to 10 times difference in latency. Thus, the large latency gap between DRAM and flash SSD makes it difficult to use the latter as a main memory extension for applications.

Our earlier work [7] investigated the potential of using flash memory as a large and slow memory for stencil computations, focusing on the case where the flash memory is used as a swap device under the *fast swap* scheme recently introduced into the Linux kernel [8][9]. In that work, a locality-aware, hierarchical out-of-core computation algorithm with data structure blocking techniques was newly employed to bridge the DRAM-flash latency divide for stencil computations. It showed that flash memory is viable as large and slow memory.

Multi-level tiling in spatial and temporal spaces, a well-known technique to optimize data access locality and parallelism, has been studied in the literature, but the algorithms designed for cache-DRAM tiers cannot be used directly for DRAM-Flash tiers because they introduce different situations under the existing spatial/temporal blocking optimizations. One major difference is the access latency gap between DRAM-Flash tiers, which is much larger than that between cache-DRAM tiers. Another difference is actual access granularity and access mechanism. The typical flash SSD is accessed in blocks, with typical size 512 B or 4 KB, as a block device, whereas cache and DRAM are accessed in finer grained cache line and word/byte as memory. Moreover, there are several access paths from an application to a Flash SSD as a block device in OS kernel layers [12], as shown in Fig. 1. Each path exhibits a different performance depending on the version of the OS kernel (Fig. 2), which has been intensively studied and advances made by incorporating new features for high-speed NVMs. In addition to the swap method, we propose two additional configurations, mmap and aio, for accessing flash media [10] [11]. The aio and mmap methods achieve more efficient access to flash than the swap method does, but there remains the possibility of achieving even better performance through more elaborate tuning.

In this paper, we compare the three methods in terms of performance with elaborate tuning in blocking data memory layout, the work-share scheme for multiple cores, and the affinity control for Non-Uniform Memory architecture (NUMA) systems. These tunings double the performance of the mmap and aio methods. As a result, using only 32 GiB of DRAM and a flash SSD, on 7-point stencil computations for a 512 GiB problem (16 times larger in size than the DRAM), 87% of the Mflops execution performance achieved using only DRAM was attained. Further, in a NUMA system, the same computation for a 1 TiB problem using only 64 GiB of DRAM and flash, 80% of the Mflops performance achieved using sufficient DRAM was attained. Through our tuning experience, we developed a runtime auto-tuning mechanism to select appropriate parameters in spatial/temporal block

sizes, a work-share scheme for multi-core, and affinity control for NUMA, by retrieving information on the underlying system hardware. This alleviates the burden on users of choosing parameters for the individual systems they utilize.

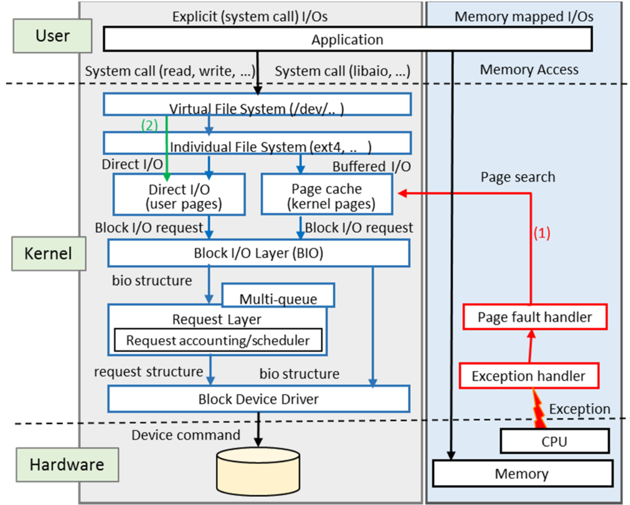

Figure 1. Multiple paths to block devices from applications

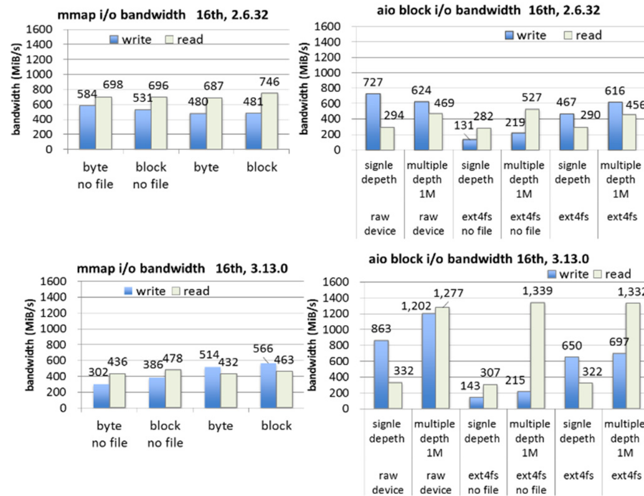

Figure 2. mmap vs. aio in IO bandwidth to flash SSD (ioDrive2) by 16 threads in kernel 2.6.32 and 3.13.0

II. LOCALITY-AWARE STENCIL ALGORITHM AND THREE METHODS FOR USING FLASH IN THE MEMORY HIERARCHY

Stencil computation is one of the most popular and important types of processing in various scientific and engineering simulations. It performs iterative calculations on a limited dataset, typically the nearest neighbor data. It sweeps all the data—e.g., three-dimensional (3D) physical data space—and updates them at each time step. In this paper, 7-point and 19-point stencil computations using the six and the 18 nearest neighbor points for a 3D data domain individually are used in general discussion for simplicity.

A. Basic temporal blocking stencil algorithm

A temporal blocking algorithm extracts not only spatial locality but also temporal locality for iterative applications. Temporal blocking optimizations have predominantly been applied to cache and DRAM tiers, the host memory, graphics processing unit (GPU) memory [13] tiers, and local and remote nodes in a cluster [14], in order to expedite data access by exploiting temporal locality. A typical temporal blocking algorithm for the 3D data domain is shown in Fig. 3.

B. Layered Blocking and three methods for accessing flash

We introduce the blocking techniques on three-layered data structures in Fig. 4, corresponding to *Buffer arrays* in flash memory, *Block arrays* in DRAM, and virtual *iBlock arrays* in L3-cache, to extract access locality [7]. Temporal blocking is applied to the flash and DRAM tiers and spatial blocking is applied to the DRAM and cache tiers.

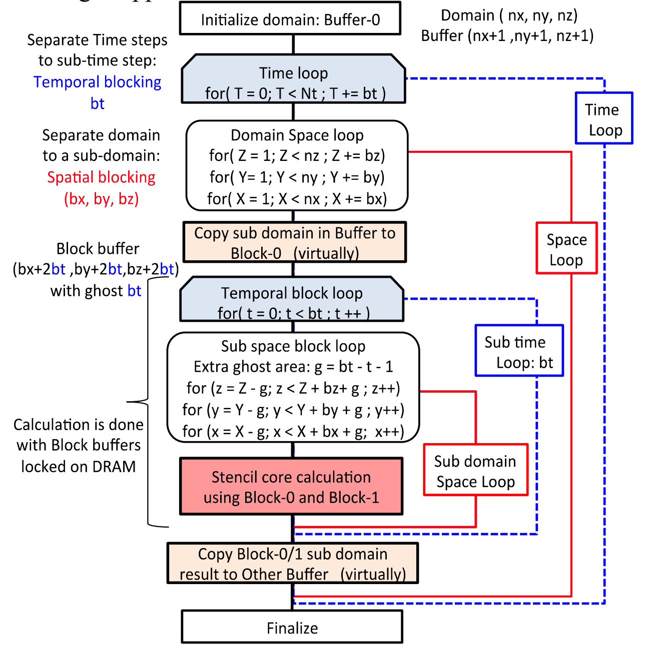

Figure 3. one-level temporal blocking algorithm: pseudo codes for a 3D domain.

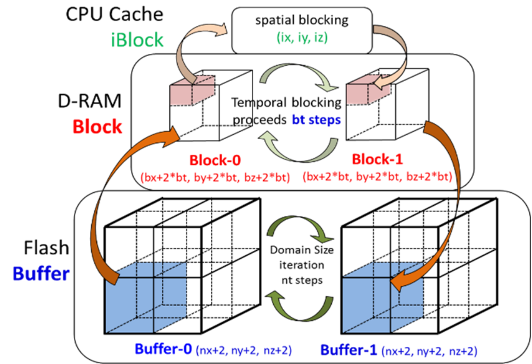

Figure 4. Three-layerd data structure for locality extraction

We propose three options for using a flash SSD as a main memory extension for application programs [10][11]: (1) swap method, (2) mmap method, and (3) aio method. In the swap method, *Block arrays* in the middle layer and *Buffer arrays* in the bottom layer in Fig. 4 are allocated by the malloc() function in applications and a flash SSD is used as a swap device under the virtual memory system of the OS. To prevent the *Block arrays* being swapped out, they are locked onto main memory by means of the mlock() function. In the mmap method, the *Block arrays* are allocated by malloc() and the *Buffer arrays* are represented as files that are memory mapped by the mmap() function in applications, and a flash SSD is used as a file system (e.g., ext4). Its access path corresponds to the arrow (1) in Fig. 2. Both methods are virtually transparent to applications.

In the aio method, Linux kernel asynchronous input/output library functions (io_submit and io_getevents) are used in the applications. The *Block arrays* are allocated by malloc() and the *Buffer arrays* are represented as consecutive blocks on a flash block device. It requires modification of application programs from memory-semantic reads/writes to explicit inputs/outputs to a flash SSD. In our experiment, a flash SSD is used as a block device and opened with O_DIRECT, corresponding to the arrow (2) in Fig. 2. It eliminates file-

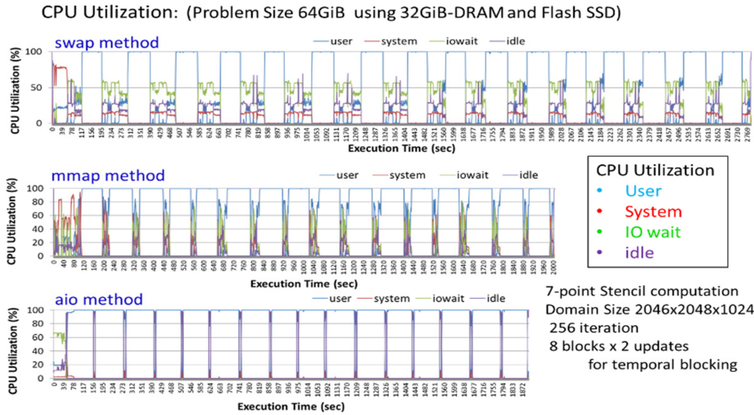

Figure 5. CPU Utilizations on three methods for 7-point stencil computation; 64GiB-problem execution on 32GiB-DRAM and a Flash

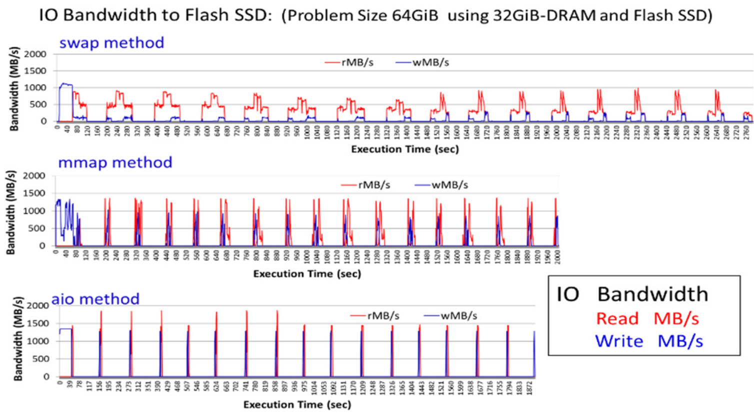

Figure 6. CPU Utilizations on three methods for 7-point stencil computation; 64GiB-problem execution on 32GiB-DRAM and a Flash

TABLE I. EXPERIMENTAL ENVIRONMENT

	UMA	NUMA
CPU	Xeon E5-2650 2.0GHz 1 socket (8 cores)	Xeon E5-2687W 3.1GHz 2 socket (8x2 cores)
Memory	DDR3-1600 ECC 8GiB x 4 (32GiB)	DDR3-1600 ECC 8GiB x 8 (64GiB)
Flash Storage (PCI Bus)	Fusion-io ioDrive2 MLC(PCIe2.0x4) 785GB, 1.2TB	
OS	kernel 3.13.0, CentOS6.4(x86_64)	
Compiler	gcc version 4.4.7 20120313 / -O3	

system-layer overhead and kernel-managed buffering, page cache. Instead, *Block arrays* in Fig. 4 are used as user page buffer and are fully controlled by applications. The recent improvement of block storage stacks in Linux, specifically, multiple IO request queues for multi-core [15], gives higher performance to asynchronous IO by multiple threads as shown in Fig. 2, but it requires block-size-aligned data access. As a result, the aio method causes complexities and restrictions in data layout in application programs.

III. PERFORMANCE IMPACT OF TEMPORAL BLOCKING ALGORITHM IN THE THREE METHODS

In this section, we explore the performance of the three configurations, swap, mmap, and aio, using a flash SSD for stencil computations as a main memory extension. The experimental setting is outlined in the UMA column of Table I.

The problem used in this preliminary evaluation was a 7-point stencil for 3D-domain data, 64 GiB problem, domain size (nx, ny, nz) $2046 \times 2048 \times 1024$ and time step iteration, Nt, 256. For the temporal blocking algorithm, the spatial blocking size (bx, by, bz) was $2046 \times 512 \times 512$ and the temporal blocking size, bt, was 128. This corresponds to two iterations of the eight *block array* calculations on 128 local iterations. The actual DRAM size used in the program was 82.2 GiB, including bt ghost areas in the *block arrays*, as shown in Fig. 4. The performance of the three methods using limited DRAM (32 GiB) and sufficient DRAM (128 GiB) was then evaluated.

Figs. 5 and 6 show CPU utilization and IO bandwidth to flash SSD profiles for each method during their execution. The 16 sections in which user CPU utilization is 100% correspond to 8-block computations in DRAM iterated twice. Each gap between these block computation parts corresponds to Input/Output from/to flash memory. Among the three methods, it can be seen that the aio method achieves the most efficient IO to flash memory and reduces its total execution time. Its peak IO bandwidth achieves almost the maximum value specified in the flash device, ioDrive2. This is as a result of the highly parallel asynchronous IO by multi-core with deep IO queue depth for only necessary data/block, eliminating the unnecessary page IO seen in swap and mmap methods.

Fig. 7 shows the relative execution times for the problems following elaborate tuning (which will be described in section V). With the aio method, the computation time for the 64 GiB problem using 32 GiB of DRAM and flash is only 1.5 times greater (1120 s) than that of the normal execution (740 s) using sufficient DRAM, 128 GiB. Without the temporal blocking algorithm, its computation time using 32 GiB DRAM is 65.2 times greater (48,232 s) than the normal execution (740 s), as shown in Fig. 7. Temporal blocking has a significant impact on the performance of the stencil computation using flash memory. The aio method with temporal blocking is most effective for execution under limited DRAM.

Fig. 8 shows the relative effective Mflops for problems of various sizes using 32 GiB with the aio method. In the 512 GiB problem, execution using only 32 GiB of DRAM achieves performance that is 87% that of normal execution using sufficient DRAM in the case of the 16-GiB problem (leftmost column in Fig. 8).

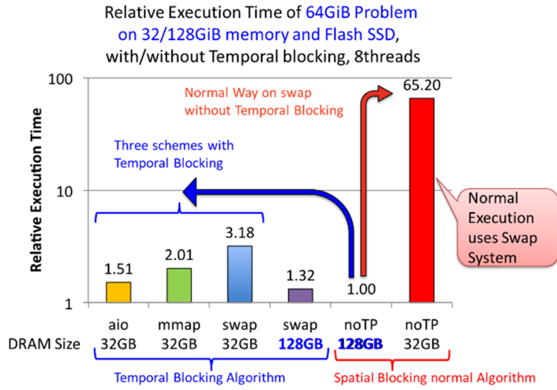

Figure 7. Relative times for various methods for 7-point stencil comp.

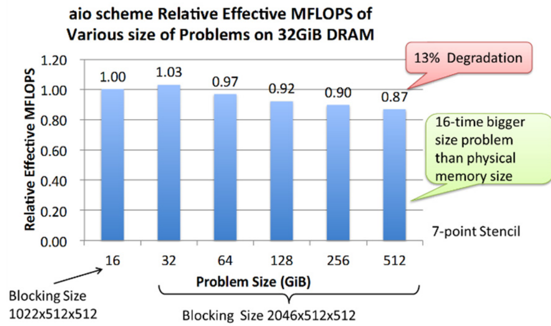

Figure 8. Performances in various-size problems on fixed physical memory (32GiB)

IV. PERFORMANCE OF AIO AND MMAP METHODS IN NUMA SYSTEMS

We introduced NUMA-aware computing in the algorithm for multiple-socket NUMA systems, as well as optimization for single-socket systems, such as the memory layout of the *Block arrays* and a work-share scheme with multi-core.

In this section, we compare the aio and mmap methods for problems of various sizes using 7-point and 19-point stencil computations in the two-socket NUMA system outlined in Table I. Fig. 9 and 10 show the execution times and the effective Mflops in the aio and mmap methods, respectively. The execution time of the aio method is 50–60% that of the mmap method. Moreover, while under execution using the mmap method, the 19-point 1 TiB problem is terminated by an out of memory (OOM) killer in the OS, because of lack of available memory in the large-size problem file mmap. In contrast, the execution of the aio method exhibits stable behavior.

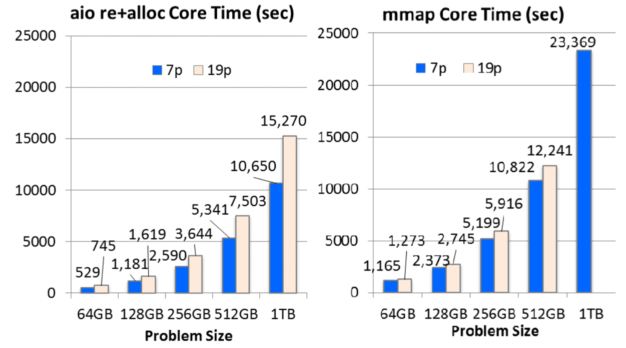

Figure 9. Execution times of various-size problems in aio and mmap methods

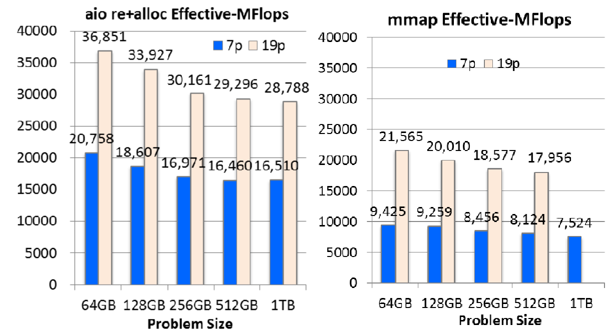

Figure 10. Effective MFlops of various-size problems in aio and mmap methods

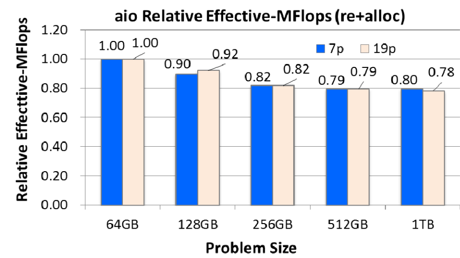

Figure 11. Relative performances in various-size problems on fixed physical memory (64GiB) in a NUMA system

Fig. 11 shows the relative Mflops in the aio method based on the performance of the execution using 64 GiB of physical memory. The 1 TiB problem execution exhibits 80% of the performance of the 64 GiB problem in 7-point stencil computation. Fig. 12 shows the comparison of the aio and

mmap methods in Mflops. The aio method achieves better performance than that of the mmap method even in the 32 GiB problems using sufficient DRAM.

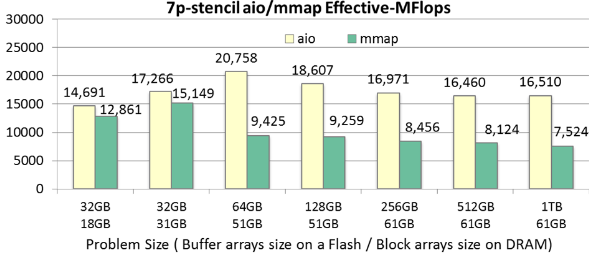

Figure 12. Effective MFlops on fixed-size memory (64GiB) in a NUMA system on aio and mmap methods

V. Optimizations in the Implementation

This section describes several performance tuning strategies used in the implementation of the algorithm for flash SSDs in section III and IV. The same 64 GiB-problem in section III was used for the evaluation on UMA systems.

A. Blocking sizes for DRAM-flash SSD tiers

The results of our previous evaluation [7] indicated that, in the temporal blocking for DRAM-flash tiers, larger spatial/temporal blocking sizes result in better performance, as long as the *Block arrays* fit inside the main memory (DRAM) capacity. The fact that the access latency of flash memory is much greater than CPU calculation cycles makes it very different from the case using a temporal blocking algorithm applied to cache-DRAM tiers [7]. The latter case usually has the best tradeoff point between increasing redundant computation overhead and speedup by exploring data access locality when choosing temporal blocking sizes. In addition to choosing a larger volume for the blocking sizes combination, bt and (bx, by, bz), it is also important to choose the shape of the array to increase sequential access in memory. For example, for the same *Block array* volume, a larger bx is better than a larger bz.

B. Memory layout of the Block arrays in the aio method

Asynchronous IO parameters, such as start-address, offset, and size, must be aligned in device block sizes—4 KB in our case. In the aio method, the *Block arrays* in Fig. 4 are implemented with a z-dimension pointer-array and multiple xy-planes pointed to by a pointer in the z-dimension array. This is in contrast to the layout in the mmap and swap methods, where they are implemented as a typical sequential C memory array. The start-address and the size of each xy-plane in the aio method are aligned in device block size. Each xy-plane is a unit comprising asynchronous IOs by multiple threads in parallel.

The performance of the three memory layouts is compared in Fig. 12. In the *individual layout*, each xy-plane is allocated with the posix_memaligned() function. Thus, the start-addresses of the xy-planes are all discrete but aligned in the device block size. In the *sequential layout,* the xy-planes are placed continuously in one sequential memory area. In the third layout, *seq+pgpad*, page (4 kB) padding is introduced in the *sequential layout*. In the current implementation of the aio method, we introduced the premise that the domain data size in x-dimension, nx, must be a multiple of the device block size. Moreover, we set the space blocking size in the x-dimension, bx, to be equal to nx for the larger IO granularity, one xy-plane in a *Block array*. Otherwise, the IO granularity becomes smaller, one x-line, for example.

C. The work-share scheme for iBlock arrays in cache

The *iBlock arrays* in Fig. 4 are virtual arrays that are used for space blocking in *Block array* calculations to increase L3 cache hits. The *iBlock* volume is determined by the size of the L3 cache and its shape is chosen to increase sequential memory access and the efficiency of the work-share schemes by the CPU cores. We explored two schemes, y-loop and z-loop parallel executions for *iBlock* with appropriate *iBlock* shapes (Fig. 14).

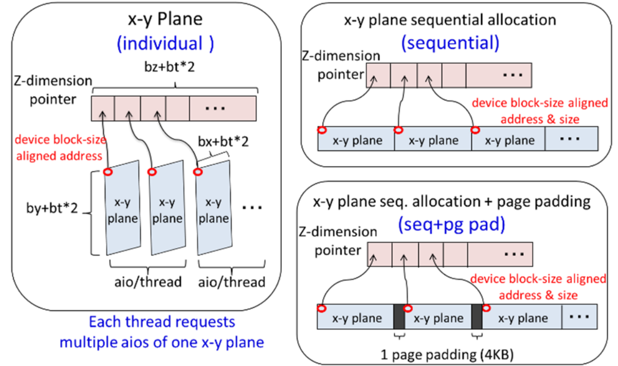

Figure 13. Three memory layouts for *Block arrays* for block-aligned access

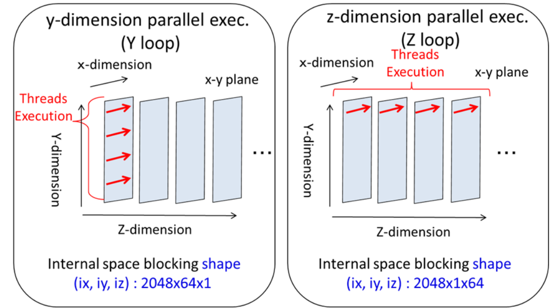

Figure 14. *iBlock arrays* spatial blocking shape and work share scheme among threads, for internal loop for L3 cache :

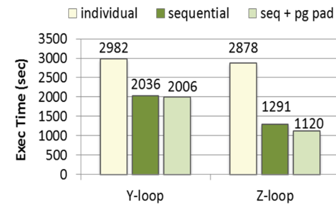

Figure 15. Left: Impact of work-share schemes for *iBlock array* and memory layouts for *Block arrays*

Fig. 15 shows the execution times in all combinations of three memory layouts and two work-share schemes. The execution times in the y-loop and the z-loop in the *sequential layout* were reduced to 70% and 50% those of the *individual layout* for respectively. In *sequential layout*, the z-loop parallel execution time was 60% of that of the y-loop parallel. The page padding was also effective in the z-loop parallel, and reduced the execution time without padding by 13%.

We also optimized the mmap method with similar strategies to those used in the blocking sizes and work-share schemes. Because the memory layout of the *Block arrays* in mmap is that of a typical C array, element padding was introduced to the arrays in the x-dimension instead of the page padding in the aio method. The optimized aio and mmap methods show reduced execution times of 55% and 59% those of the methods without optimization, respectively, as shown in Fig. 16.

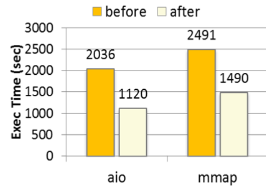

Figure 16. Before and after of optimizations in aio and mmap methods

D. Data-core affinity control for NUMA systems

In a NUMA system with *n* CPU-sockets, *Block arrays* are virtually divided, along the z-dimension, into *n* sub-blocks that are calculated in each CPU-socket with local cores, as shown in Fig. 17. This is carried out by OpenMP parallel sections, each of which calls sched_setaffinity() for thread-CPU binding. A sub-block and a CPU-socket binding is also carried out by mbind() when *Block arrays* are initially allocated, or by repeated calls of malloc() and free() at every *Block array* calculation. These NUMA-aware tunings achieve a 55% increase in effective performance (Mflops) compared to the performance without the tunings.

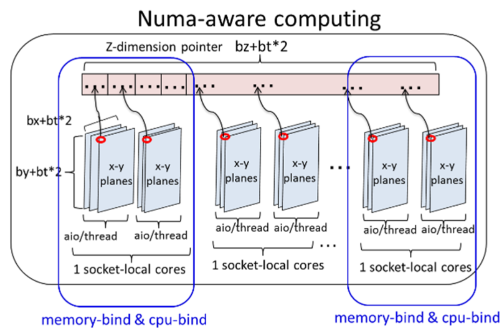

Figure 17. Data layout and affinity-control for NUMA systems

E. Runtime Auto-tuning mechanism

Using the above strategies, we implemented an automatic tuning mechanism for temporal and spatial blocking parameters for each memory layer by extracting underlying hardware information, such as DRAM/cache size, number of cores/sockets, and device capacity/block-size. The mechanism also incorporates NUMA-aware computations automatically, and uses Portable Hardware Locality (hwloc) [17] to retrieve the system hardware information. With this mechanism, users can easily run the stencil programs using the aio method simply by specifying the domain size (nx, ny, nz), time steps (Nt), and the path to a flash device, such as "./stencil7p -n 4094 4096 2048 -t 1000 -d /dev/sdc."

VI. Conclusions

In this paper, we investigated the performances in three different configurations of stencil computation to access a flash device as main memory extension after elaborated tuning. We found that aio method gained the highest performance but it generated restrictions in data layout in user programs. In contrast, mmap method is easier to use and the access to a flash is transparent for users, but its performance is limited about 50%-60% of that of aio method.

We are currently extending this configuration for using various memories in vertical and horizontal directions, GPU memory and hard disks in one node and remote nodes over a cluster system. The future work also includes development of more general APIs for various kinds of stencil computations.

References

[1] Anirudh Badam, "How Persistent Memory Will Change Software Systems", IEEE Computer , pp45-51, Aug. 2013

[2] Persistent Memory Programming http://pmem.io

[3] Kshitij Sudan, Anirudh Badam, Dvid Nellans, "NAND-Flash: Fast Storage or Slow Memory?", NVM Workshop 2012

[4] Brian Van Essen, et.al "DI-MMAP—a scalable memory-map runtime for out-of-core data-intensive applications", Cluster Computing , Vol.18, Issu1, pp.15-28, March 2015

[5] SanDisk ioDrive2, http://www.sandisk.com/enterprise/pcie_flash/ fusion-iomemory-iodrive2/

[6] Intel SSD DC P3700, http://www.intel.com/content/www/us/en/solid-state-drives/intel-ssd-dc-family-for-pcie.html.

[7] Hiroko Midorikawa, Hideyuki Tan and Toshio Endo:"An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations", Proc of the 2014 International Conference on High Performance Computing and Simulation (IEEE HPCS2014), pp.268-277, July 2014

[8] Improve Linux swap for High speed Flash Storage http://events.linuxfoundation.org/sites/events/files/lcjpcojp13_shaohua.pdf., The Linux Foundation, 2013

[9] OpenNVM, FusionIO, http://opennvm.github.io.

[10] Hiroko Midorikawa, "Using a Flash as Large and Slow Memory for Stencil Computations". Flash Memory Summit 2014, Aug. 2014

[11] Hiroko.Midorikawa, "Using a Flash SSDs as Main Memory Extension with a Locality-aware Algorithm". 2015 Non-Volitile Memories Workshop, in UCSD, March 2015

[12] Linux Storage Stack Diagram kernel 3.17, https://www.thomas-krenn.com/de/wikiDE/images/2/24/Linux-storage-stack-diagram_v3.17.pdf

[13] Guanghao Jin, Toshio Endo and Satoshi Matsuoka, "A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs", IEEE Cluster2013, 2013

[14] M. Wittmann, G. Hager, and G. Wellein, "Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory", Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages, April 2010

[15] M Bjørling, J Axboe, D Nellans, P Bonnet,"Linux block IO: introducing multi-queue SSD access on multi-core systems", Proc.of the 6th International Systems and Storage Conference (SYSTOR '13)

[16] "Linux Block IO: Introducing Multiqueue SSD Access on Multicore Systems", M. Bjorling et.al, 2013, http://bjorling.me/blkmq-slides.pdf

[17] Portable Hardware Locality http://www.open-mpi.org/projects/hwloc/