An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations

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

Toshio Endo
Global Scientific Information and Computing Center
Tokyo Institute of Technology, JST CREST
Tokyo, Japan
endo@is.titech.ac.jp

Abstract—**This paper investigates the potential of flash as a large and slow memory behind dynamic random-access memory (DRAM) for stencil computation, which is one of the most common and important computation kernels in various scientific and engineering simulations. We evaluate the performance of a fastswap kernel, which was recently incorporated into Linux, in stencil computation using flash as a swap device. Moreover, we propose a locality-aware, hierarchical out-of-core computation algorithm by employing data structure blocking techniques in stencil computations to bridge the DRAM-flash latency divide. We find that 7-point and 27-point stencil computations for a 1-TiB problem size (32 times that of the DRAM), using only a 32-GiB DRAM and a flash solid-state drive (SSD), in Mflops attain 24% and 47%, respectively, of the performance achieved in execution using only DRAM.**

Keywords—Non-volatile memory (NVM); flash memory; SSD; memory hierarchy; temporal blocking; stencil computation; slow memory; access locality; locality-aware.

I. INTRODUCTION

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 data set, typically the nearest neighbor data. It sweeps the entire data – e.g., three-dimensional (3D) physical data space – and updates them at each time step. Fig. 1 shows a typical stencil computation for a 3D data grid – a 7-point stencil computation using the six nearest neighbor points and a 19-point stencil computation using the 18 nearest neighbor points.

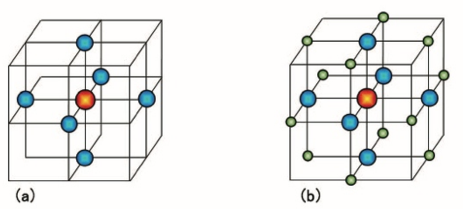


Fig. 1. 7-point and 19-point stencil calculations on 3D data.

Like most scientific computation, stencil computation often requires a large memory to tackle big-sized problems or for higher resolution data analysis. However, there is a limit to the extent to which dynamic random access memory (DRAM) can be increased in main memory, because there is a limited number of memory slots on server boards, limited power consumption, and other resource limitations. The most direct and the traditional way to use data larger than the size of the physical memory is through virtual memory system in an operating system (OS), where the program implicitly uses a swap device instead of the main memory. However, in modern high-performance computing, it is common practice to avoid paging to a swap device that is prohibitively slow with a hard disk drive (HDD). The swap system in an OS kernel, which was originally designed for slow HDDs and small memory, and has not been significantly upgraded for a long time. This is a reason for why paging to a swap device becomes obsolete in high-performance computing.

On the other hand, the advent of new non-volatile memories (NVMs) influences not only the traditional memory hierarchy, but also the basic idea of memory read/write and file input/output (IO) in traditional programming models [1]. Many kinds of NVMs, such as flash memory, Resistive Random-Access Memory (ReRAM), Phase-Change Memory (PCM) and Magnetoresistive RAM (MRAM) are being extensively researched and developed nowadays. Of these NVMs, flash memory is already widely available to end-users. Its access time is not as short as that of DRAM, but it provides a much greater capacity at a lower cost and with less power consumption. It is already used as a front-end cache in large hybrid storage systems with HDDs and an intermediate tier between the DRAM memory and the HDDs. However, in the most cases, a flash is still employed as a storage device and used through file input/output (IO) function calls from applications. With regard to power consumption, cost and space, flash memory is a natural choice as a DRAM extension to the main memory.

There are several options in the use of flash solid state drives (SSD) as memory, e.g., using a file memory map mechanism with the existing *mmap()* function in Unix, or using new memory allocation functions specially designed for flash SSDs, such as *SSDAlloc()* [4][5]. Of these, the most convenient and transparent way for users is to use a flash SSD implicitly as a swap device under the virtual memory system of the OS. Two improvements have recently been made in use of flash SSDs as a swap device. One is the development of Peripheral Component Interconnect Express (PCIe) bus-connected flash SSDs [2][3]. The access latency levels of these are several hundred times lower than those of HDDs. The other improvement is the introduction of a revised swap system for the OS kernel: fastswap [6][7][8]. It is designed for high-speed swap devices, such as a flash SSD.

The access latency of PCIe-based flash SSDs is much smaller than that of HDDs, but its latency level is several hundred to a thousand times greater than the latency of DRAMs, as shown in Fig. 1. The gap in latency between DRAM and flash SSD is much larger than that between the level-3 (L3) cache and main memory. The L3 cache is only three to 10 times faster than the main memory. Thus, the large latency gap between DRAM and flash SSD makes it difficult to use the latter as a main memory extension for applications.

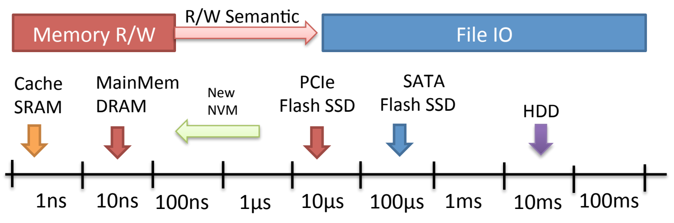


Fig. 2. Latency in various devices.

In this backdrop, we first investigate several aspects of flash SSDs as a main memory extension in the new swap kernels. We then design and introduce a locality-aware algorithm to a stencil computation. The stencil computation has desirable levels of spatial and temporal locality in data access. A standard optimization procedure such as spatial blocking is usually applied to obtain optimal spatial locality in memory access. Moreover, temporal blocking optimization is also applicable to regular data access patterns in iterative update procedures. Temporal blocking involves updates of several time steps for a local small block before proceeding to the next block. It boosts data access speed by exploiting temporal locality. In this paper, we apply this optimization to the DRAM and flash SSD tiers in stencil computations and investigate the potential of flash SSD as a large and slow memory for stencil computations.

Our contributions are summarized as follows:

- The basic performance of a PCIe-based flash SSD as a swap device is investigated under the widely used conventional kernel and compared with a HDD and an ordinary flash SSD. Moreover, the impact of the fastswap kernel in using a PCIe-based flash SSD is evaluated for the STREAM benchmark [9] and for stencil computations.
- Under the fastswap kernel, a hierarchical out-of-core computation algorithm is newly designed and applied in a stencil computation to bridge the DRAM-flash latency divide. It performs sufficiently well for practical use, thus exhibiting the potential of a flash SSD as slow and large memory.

In section II, we conduct preliminary performance evaluations by comparing PCIe-based flash SSD as a swap device with a HDD and a conventional flash SSD through STREAM benchmark [9]. In section III, we evaluate the effect of the fastswap kernel on a PCIe-based flash SSD for STREAM benchmark and stencil computations. In section IV, we propose a spatial and temporal blocking optimization algorithm, which designed for the memory tier of the DRAM and flash. Section V presents a preliminary parameter tuning for the temporal blocking for a flash SSD. In Section VI, we test the performance of stencil computing with multi-level optimization, temporal and spatial blocking, when using flash SSD as slow memory. In section VII, we outline our conclusions and directions for future work.

II. SWAP DEVICE PERFORMANCE UNDER TRADITIONAL KERNEL

In this section, we investigate the basic performance of PCIe-based flash SSDs as a swap device in comparison with a serial advanced technology attachment (SATA)-based HDD and a SATA-based flash SSD under the conventional CentOS6 (kernel 2.6.32). The experimental setting is shown in Table I.

TABLE I. EXPERIMENTAL ENVIRONMENT 1

CPU	Xeon E5-2687W 3.10GHz x 2 (16cores)
Memory	32 - 128 GiB (Max Total Mem: 128GiB) 8GiB(DDR3 1600MHz ECC Reg) x 16
OS	CentOS6 (2.6.32-279.14.1.el6.x86_64)
Compiler	gcc 4.4.6 -O3
SwapDevice	ioD / SSD / HDD

	Product	capacity	Interface
ioD	ioDrive2 MLC (FusionIO)	1.2TB	PCIe 2.0 x 4
SSD	A1 SSD (innodisk)	240GB	SATA3
HDD	WD1003FBYX-0(WDC)	1TB	SATA

Fig. 3 compares the execution times of the STREAM benchmark [9] in three cases: 1) using SATA3-baed HDD, 2) using traditional SATA-based flash SSD, and 3) using PCIe-based flash device (FusionIO ioDrive2 (ioD)) [2], as a swap device. This experiment is conducted on a fixed-size physical memory of 32 GiB with a varying number of elements in arrays used in the STREAM benchmark. STREAM is designed to measure the bandwidth at each level of the memory hierarchy by changing the size of arrays used inside. It uses three arrays in four calculation types (COPY: $a(i) = b(i)$, SCALE: $a(i) = q \times b(i)$, SUM: $a(i) = b(i) + c(i)$, TRIAD: $a(i) = b(i) + q \times c(i)$). It scans entire arrays sequentially for each calculation. Thus, its memory access locality is low and floating operations take up a small fraction of the entire execution time.

The horizontal axis in Fig.3 represents a ***physical memory ratio***, the ratio of used physical memory to the program's virtual size. The drop in the *physical memory ratio* from 100% to 24.5% on the horizontal axis corresponds to the virtual memory size of the array data from 2.3 GiB to 109.9 GiB. The vertical axis represents the relative execution time of STREAM data running on the physical memory. These are individually normalized by the time taken for the execution of each set of data using the full 128-GiB memory. The leftmost line represents the case where a HDD is used as a swap device, which significantly increases the execution time as the array size increases. Processing 1,600M array elements, corresponding to 37GiB, on a 32-GiB physical memory server takes about six hours (21,440 s). The original execution time using full memory is only 147 sec. For the same amount of processing, SSD takes 3.3 hours (12,150 sec) whereas the ioD takes 1.7 hours (6,381 sec). The HDD is prohibitively slow, but ioD relative time saturated to 70–94 times larger value compared to on-memory processing time. Fig. 4 shows the relative execution times of arrays 32GiB and 37GiB in size, normalized by the execution times on full memory. Even when processing a 32-GiB array, the process cannot use all the physical memory, and thus execution times rapidly increase.

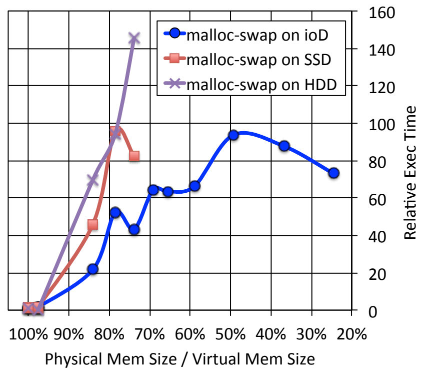


Fig. 3. STREAM: Relative time for data larger than physical memory size (32 GiB, 1 thread, CentOS6 kernel 2.6.32).

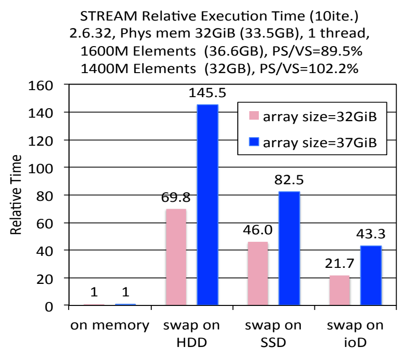


Fig. 4. Relative times of STREAM, with 10-iterations for array data size 32.0GB and 36.6GB on 32 GiB physical memory, 1 thread, CentOS6 kernel 2.6.32

III. THE IMPACT OF FASTSWAP KERNEL PERFORMANCE

In this section, the impact of the fastswap kernel in using a PCIe-based flash SSD is evaluated. Our evaluation uses ioDrive2 (ioD) [2], as swap devices for three Linux kernels – 2.6.32 (CentOS6), 3.6.0 and fastswap, which is the patched version of 3.6.0 incorporated with nvm-fastswap [7][8]. The experimental environment is shown in Table II.

TABLE II. EXPERIMENTAL ENVIRONMENT 2

CPU	Xeon E5-2650 2.00GHz x1 (8cores)
Memory	32GiB boot (Total memory 128GiB) 16GiB(DDR3 1600MHz ECC Reg) x 8
OS kernel	CentOS6 (2.6.32)
	3.6.0
	fastswap (3.6.0 + nvm-fast-swap)
Compiler	gcc 4.4.7 20120313 -O3
SwapDevice	ioDrive2 (FusionIO)

A. *STREAM benchmark under Fast Swap and other Kernels*

The performances of STREAM benchmark under the three kernels are investigated. Fig. 5 and Fig. 6 show the execution time and the average bandwidth, when the size of STREAM array is 36.6GiB, 1600M elements, and physical memory size is 32GiB. The measured values of STREAM sometimes fluctuate when a swap daemon is running, but kernel 3.6.0 has a poor performance for multi threads executions in this experiment.

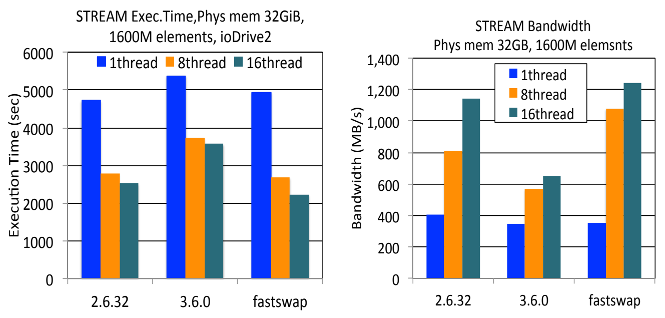


Fig. 5. STREAM Performance on various kernels, 1600M elements, 10 iterations, 1-16 threads, ioDrive2, two-socket server in Table I

B. *A Stencil Computation under Fast Swap and other Kernels*

We now evaluate the stencil computations with ioD under the three kernels. Fig. 6 shows the core execution time, which includes iterative steps and excludes data initialization, for the 7-point stencil computation shown in Fig. 1. The 3D grid domain size used here is 2048 x 2048 x 1024, which corresponds to 64 GiB and is two times larger in size than physical memory. This stencil computation employs a standard 3D-space blocking optimization to increase data access locality. The sub-block size for a space blocking is 32 x 32 x 32. It is the size that fits into L3 cache. In this experiment, the number of iterations is 256. According to Fig. 6, there is not much of a difference between the fastswap and the 3.6.0 kernel. The execution times on 32-GiB physical memory are 12.3 times longer than that using 128-GiB memory under the fastswap. Although the physical memory ratio is about 50%, its performance is not so poor compared to the STREAM case shown in Fig. 3. This is because a stencil computation contains more memory access locality than the STREAM benchmark. However, the performance described above is not satisfactory for flash memory as a main memory extension. Thus, we introduce a temporal blocking algorithm to increase data access locality in stencil computations.

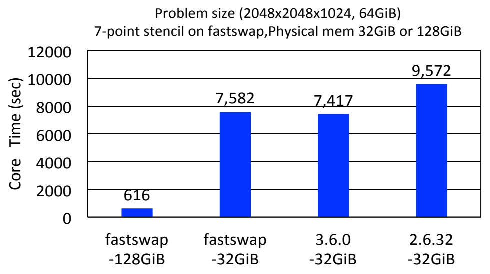


Fig. 6. Kernel comparison of 2048x2048x1024, 7-point stencil (64GiB problem), 256 iterations, on a 32-GiB physical memory and ioD as swap device with *madvise*, 8 threads, one-socket server

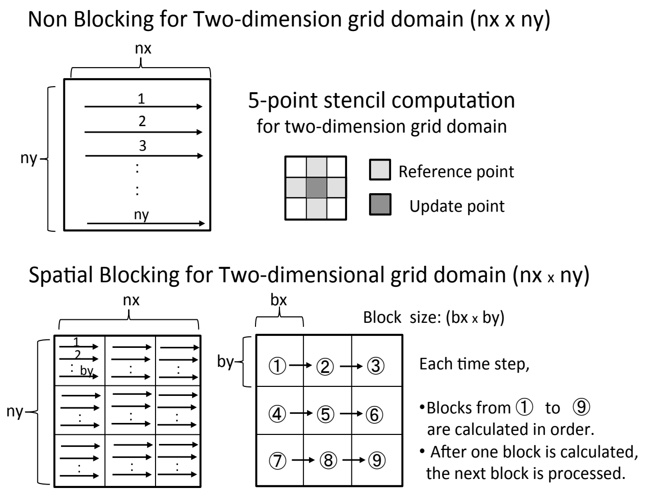
IV. A LOCALITY-AWARE ALGORITHM FOR USING FLASH SSD AS SLOW MEMORY

In this section, we introduce a locality-aware, hierarchical out-of-core computation algorithm that uses spatial and temporal blocking optimization. Our algorithm is incorporated into stencil computation to bridge the large flash-DRAM latency gap when flash is used as main memory extensions.

A. A Temporal Blocking Optimization

Temporal blocking involves updating several time steps for a local small block before proceeding to the next block. Here, we use a two-dimensional grid array and 5-point stencil computation to explain temporal blocking optimization.

Spatial blocking optimization typically extracts spatial access locality, and its algorithm is shown in Fig. 7. Each spatial block computation reads grid data from the reference area and writes results to the update area, as shown in Fig. 8.



Non Blocking for Two-dimension grid domain (nx x ny)
nx
ny
1
2
3
ny
5-point stencil computation
for two-dimension grid domain
Reference point
Update point
Spatial Blocking for Two-dimensional grid domain (nx x ny)
nx
bx
by
ny
Block size: (bx x by)
① → ② → ③
④ → ⑤ → ⑥
⑦ → ⑧ → ⑨
Each time step,
•Blocks from ① to ⑨ are calculated in order.
• After one block is calculated, the next block is processed.

Fig. 7. Non-blocking and spatial blocking for stencil computations

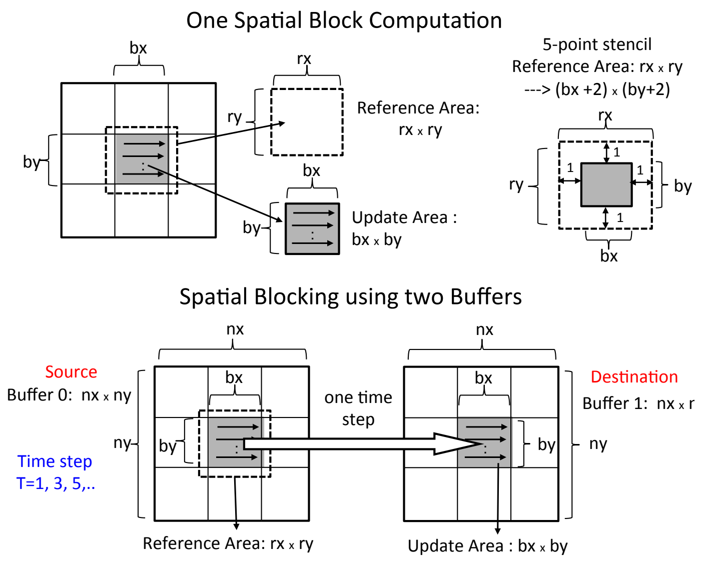
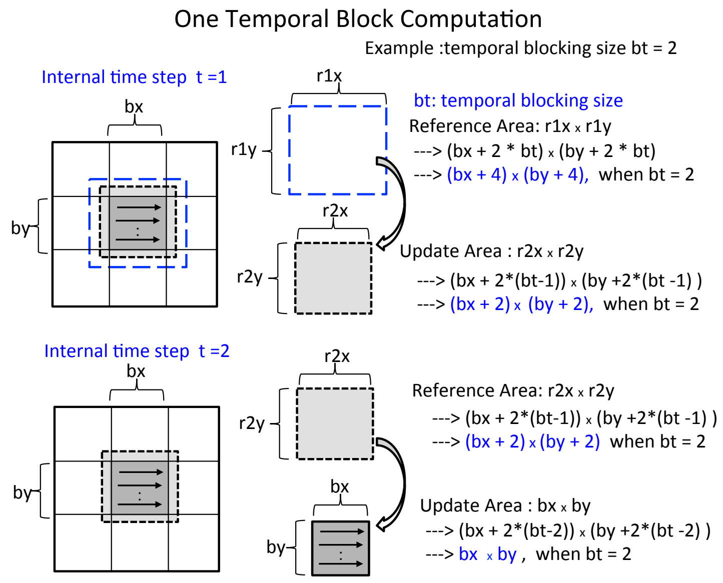


Fig. 8. Spatial blocking using two buffers: every time step, source and destination buffers are exchanged.

In conventional stencil computation, grid data are stored in one of two buffers – the source buffer – and data in the source buffer are read, calculated, and written to the other buffer – the destination buffer. After updating the 3D data, which constitutes one time-step iteration, the source and the destination buffers are exchanged.



One Temporal Block Computation
Example :temporal blocking size bt = 2
Internal time step t =1
bx
by
r1x
r1y
bt: temporal blocking size
Reference Area: r1x x r1y
---> (bx + 2 * bt) x (by + 2 * bt)
---> (bx + 4) x (by + 4), when bt = 2
r2x
r2y
Update Area : r2x x r2y
---> (bx + 2*(bt-1)) x (by +2*(bt -1))
---> (bx + 2) x (by + 2), when bt = 2
Internal time step t =2
Reference Area: r2x x r2y
---> (bx + 2*(bt-1)) x (by +2*(bt -1))
---> (bx + 2) x (by + 2) when bt = 2
Update Area : bx x by
---> (bx + 2*(bt-2)) x (by +2*(bt -2))
---> bx x by , when bt = 2

Fig. 9. One temporal block computation: It advances bt time steps internally. It reads data in reference area (bx + 2 × bt) × (by + 2 × bt) and finally updates the data in update area: bx × by.

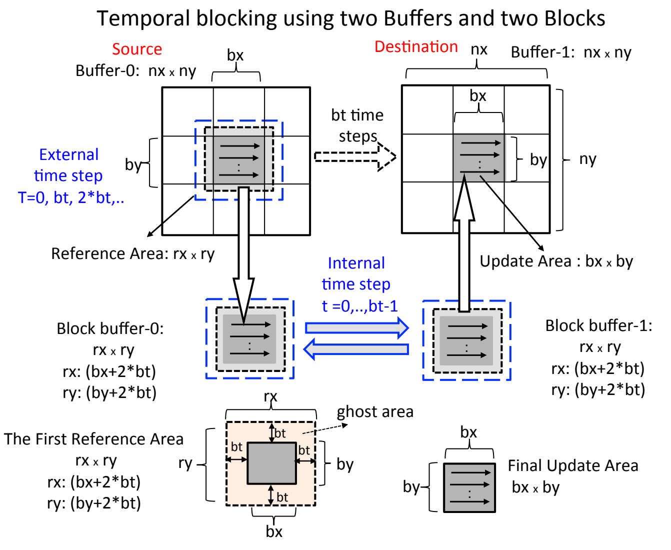


Fig. 10. Temporal blocking using two buffers and two block buffers.

On the other hand, temporal blocking optimization extracts both spatial and temporal access localities. Temporal blocking divides the entire time space into sub-time blocks for iterative computations. One temporal block computation updates all grid data in a spatial block in several time steps. The number of time-steps advanced locally is temporal block size (bt). Fig. 9 shows the example of one temporal block computation when bt equals 2. In this case, one block computation updates data two times to advance two time steps, before proceeding to the next block computation. Every temporal blocking update requires a larger update area than the original spatial block size, in order to locally advance the time step. Thus, another two block buffers have to be prepared in addition to the two buffers, as shown in Fig. 10. Using these two block buffers, one temporal blocking computation, bt-times updates , is carried out. After one temporal block computation is finished, the result in the block buffer is copied back to a destination buffer.

B. *A Temporal Blocking Algorithm for DRAM and Flash*

The temporal blocking algorithm of flash SSD and the memory is shown in Fig. 11 and Fig. 12. In this algorithm, two block buffers are prepared in addition to two 3D data buffers. The data buffers are divided into several 3D blocks. In the first iteration, one block part of the buffer data – e.g., Buffer-0 – is calculated as source data, and the result data are written to a destination block buffer – e.g., Block-0. In the second iteration, the source data in Block-0 are calculated, and updated data are written to the other block buffer – Block-1. In all iterations except the first and the last, the two block buffers are used in updating calculations by exchanging source and destination. At the final iteration, the block buffer written to last becomes the source buffer, while the other data buffer, Buffer-1, becomes the destination buffer.

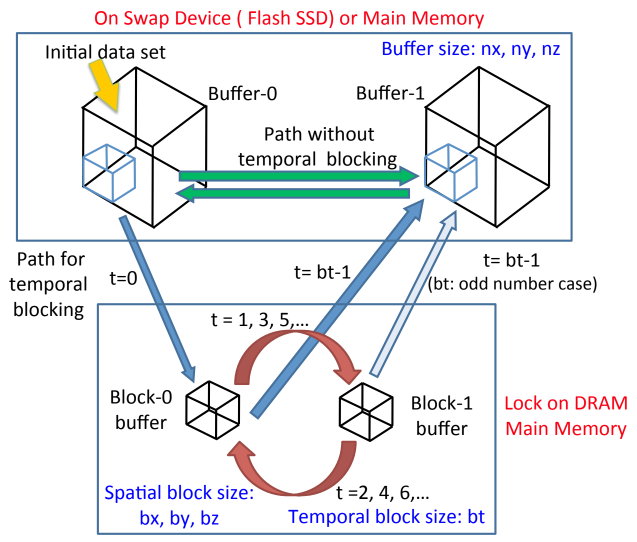


Fig. 11. Calculating data flow in temporal blocking for flash and DRAM tiers.

In this experiment, two 3D data buffers and blocks are allocated by *malloc()* with MADV_SEQUENTIAL *madvise()*. The size of the data is larger than that of the DRAM memory, because of which most of the data are swapped out to a swap device – the flash SSD. When a program accesses the data in a swapped-out page, a swap daemon swaps in the required page from the swap device. On the other hand, block buffers are fixed on the DRAM memory by *mlock()* to prevent them from swapping out to a swap device. At the first and last iterations, update calculations access the buffer data, which are supposed to exist in the flash SSD. However, in the remaining iterations, the calculations only use data on the block buffers in the DRAM. Actual block buffer sizes are larger than the subdomain sizes – bx, by, and bz – shown in Fig. 12. They have extra ghost regions on both sides for each dimension in 3D for temporal blocking. The size along the x-axis becomes bx + 2 × bt when the temporal block size is bt. This algorithm is called a one-level temporal blocking algorithm in this paper.

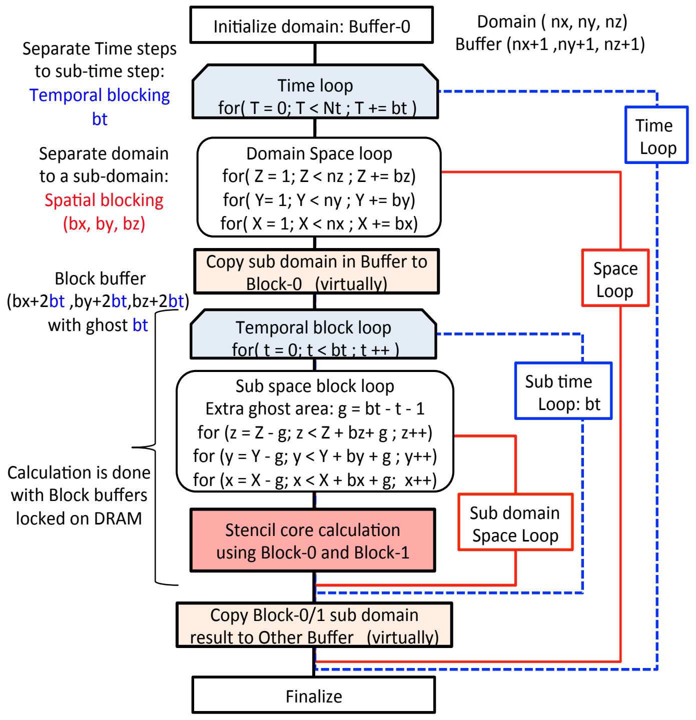

Initialize domain: Buffer-0
Domain (nx, ny, nz)
Buffer (nx+1 ,ny+1, nz+1)
Separate Time steps
to sub-time step:
Temporal blocking
bt
Time loop
for(T = 0; T < Nt ; T += bt)
Time
Loop
Separate domain
to a sub-domain:
Spatial blocking
(bx, by, bz)
Domain Space loop
for(Z = 1; Z < nz ; Z += bz)
for(Y= 1; Y < ny ; Y += by)
for(X = 1; X < nx ; X += bx)
Copy sub domain in Buffer to
Block-0 (virtually)
Space
Loop
Block buffer
(bx+2bt ,by+2bt,bz+2bt)
with ghost bt
Temporal block loop
for(t = 0; t < bt ; t ++)
Sub time
Loop: bt
Sub space block loop
Extra ghost area: g = bt - t - 1
for (z = Z - g; z < Z + bz+ g ; z++)
for (y = Y - g; y < Y + by + g ; y++)
for (x = X - g; x < X + bx + g; x++)
Calculation is done
with Block buffers
locked on DRAM
Stencil core calculation
using Block-0 and Block-1
Sub domain
Space Loop
Copy Block-0/1 sub domain
result to Other Buffer (virtually)
Finalize

Fig. 12. 1-level temporal blocking algorithm: pseudo codes for a 3D domain.

V. PRELIMINARY PARAMETER INVESTIGATION OF TEMPORAL BLOCKING FOR FLASH

In temporal blocking for stencil computations, there is a tradeoff between slow memory access and redundant calculation overhead for ghost areas, the size of which varies with the temporal block size. There are other tuning parameters, such as spatial blocking size versus temporal blocking size under limited DRAM memory. In this paper, using flash as slow memory is the biggest factor affecting temporal blocking for a stencil computation in comparison with other factors such as caches, NUMA memory and CPU architectures. Incorporating various levels and kinds of parameter tuning dependent on the underlining architecture generates a large parameter space to sweep, which makes results more specific and, hence, complex to analyze. Thus, in this experiment, we only apply basic parameter tuning to temporal blocking.

TABLE III. EXPERIMENTAL ENVIRONMENT 3

CPU	Xeon E5-2650 2.00GHz x1socket (8cores)
Memory	128GiB, 16GiB(DDR3 1600MHz ECC Reg) x 8
OS kernel	fastswap (3.6.0 + nvm-fast-swap)
SwapDevice	ioDrive2 (FusionIO)

To find adequate temporal and spatial blocking parameters for stencil computation on flash, a 7-point stencil is used. The experiment is performed using ioD under fastswap, as shown in Table III. The domain of a stencil computation is 3D 2048 x 2048 x 1024 double data. The total program size becomes larger than 64 GiB because it requires two spatial block buffers with an extra ghost area depending on temporal blocking size.

A. Space Block Size vs. Temporal Block Size

First, several temporal and spatial blocking size combinations are investigated on a 128-GiB full memory execution. Fig. 13 shows an *effective Mflops*, calculated by the measured core computation time, the amount of data update and the number of floating operations per data update. Each line represents subdomain spatial size, from 32 x 32 x 32 to 1024 x 1024 x 1024. This clearly shows the tradeoff between spatial and temporal block sizes. These parameter sets boost the performance by 40%-50% more than the method that uses only spatial blocking, corresponding to the leftmost point in Fig. 13.

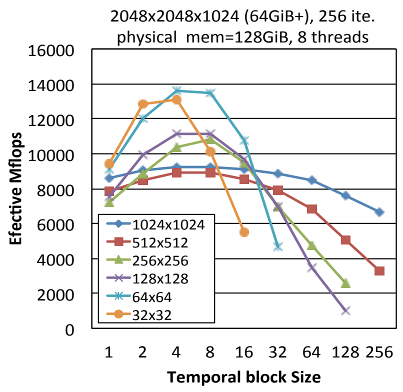


Fig. 13. Spatial block size vs. temporal block size on full memory (128GiB) 7-point stencil (2048 x 2048 x 1024 double 64 GB+, 256 item. 8 threads).

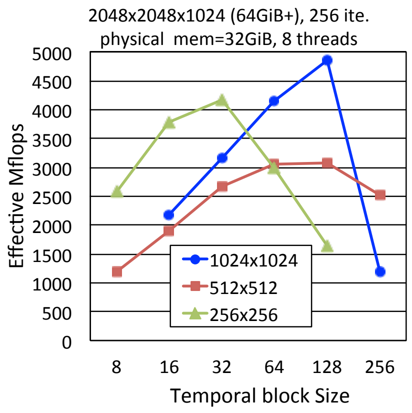


Fig. 14. Spatial block size vs. temporal block size on limited memory (32GiB) and flash 7-point stencil (2048 x 2048 x 1024 double 64 GB+, 256 item. 8 threads).

Following this, the same stencil program is evaluated on a 32-GiB physical memory, which corresponds to half of the problem size. The experiment using three spatial blocking sizes is shown in Fig. 14. The spatial size 1024 x 1024 x1024 seems to enhance performance for temporal size larger than 128 time-steps. However, the total buffer and block size with a temporal block size of 256 is 118 GiB, and the performance sharply drops because of the lack of the physical memory, as shown in Fig.14. As a result, the best performance parameter set for this problem size with 256 iterations is a spatial block size of1024 x 1024 x 1024 and a temporal block size of128 time steps.

In the flash and DRAM tiers, the cost of redundant calculations caused by temporal blocking is small compared to the access overhead due to flash. The lesson here is that the biggest spatial and temporal block size combination that can fit the available physical memory is preferable for temporal blocking for DRAM and a flash.

B. Multi-level Blocking: Temporal and Spatial Blocking

According to our previous parameter tuning result, a large spatial block size, such as 1024 x 1024 x 1024, is preferable. However, from the perspective of cache utilization, the straightforward calculation of 1024 x 1024 x 1024 is inefficient. Hence, another spatial blocking – a 32 x 32 x 32 sub-block for the 1024 x 1024 x 1024 – is introduced as a two-level temporal and space blocking. Fig. 15 shows execution times with full memory (128 GiB). Its relative times are normalized by the time taken by a one-level blocking with eight threads. Using eight threads, the two-level blocking is 2.2 times faster than one-level blocking. Thus, this two-level algorithm is used in the following experiments.

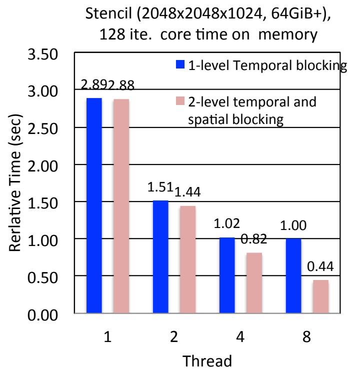


Fig. 15. One-level blocking vs. two-level blocking on full memory (128GiB), 7-point stencil (2048 x 2048 x 1024 double 64 GB+, 256 item. 8 threads), (temporal block size = 128, spatial: 1st block size = 1024, 2nd block size = 32).

VI. PERFORMANCE OF HIERARCHICAL BLOCKING FOR STENCIL COMPUTATIONS ON FLASH

In this section, we measure the performance of a two-level blocking algorithm with multiple threads for a stencil computation using limited memory and flash SSD. In this two-level blocking algorithm, temporal blocking is applied to the DRAM main memory and the flash tier, whereas the second level spatial blocking is applied to the cache and the main memory tier. A set of spatial and temporal blocking sizes is selected by using our results in the previous section by choosing the largest power of two as the spatial block size that fits in the physical memory, and the maximum temporal block size from the available combinations.

A. Varing Physical Memory size for Fixed Problem size

First, we investigated the performance of the two-level blocking algorithm with eight threads by fixing the problem size to a 2048 x 2048 x 1024 stencil problem and varying the physical memory sizes – ranging from 128 GiB to 8 GiB – in the server, as shown in Table III. Fig. 16 shows the relative execution times in both cases for 7-point and 19-point stencil computations. Fig. 17 shows *effective Flops*. The horizontal axis represents the physical memory size used for the experiment. The actual program data size is larger than the problem size as shown in Table IV. For example, at the point in Fig. 16 where the physical memory is 64 GiB, the actual program data size is 95.5 GiB and the actual physical memory ratio is 0.67 (= 64/95.5). At a physical memory size is 16 GiB, the execution times for 64GiB problem are 2.18 and 1.65 times higher, respectively, than ordinary full memory execution. The 19-point computation has more calculations per memory and cache access than the 7-point computation. Thus, the degradation in performance is smaller even with limited memory.

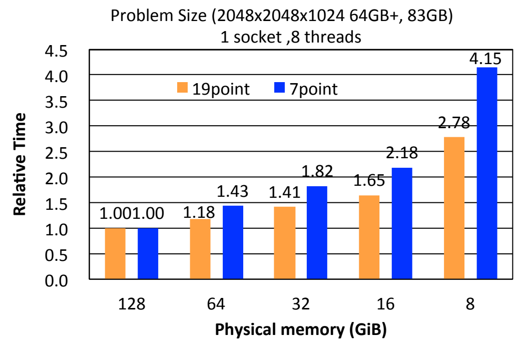


Fig. 16. Relative time for stencil with 2-level temporal blocking (2048 x 2048 x 1024 double 64 GB+, 256 ite. one-socket server, 8 threads).

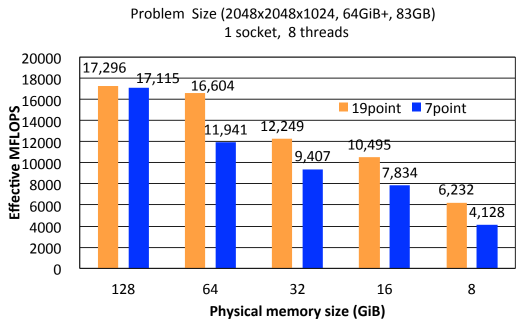


Fig. 17. Effective MFLOPS of stencil with 2-level temporal blocking (2048 x 2048 x 1024 double 64 GB+, 256 ite. one-socket server, 8 threads).

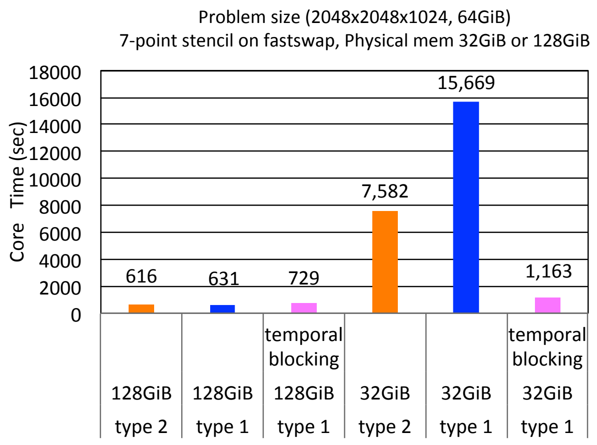
TABLE IV. PROBLEM SIZE AND ACTUAL DATA SIZE AND PHYSICAL MEMORY RATIO

	Spatial	1024x1024x1024		1024x1024x512	
	Temporal	96		128	
Problem Size	2 Buff Size (GiB)	Actual Size(GiB)	Ratio	Actual Size(GiB)	Ratio
1024x1024x1024	16*	25.1	1.57	27.3	1.71
2048x1024x1024	32	59.0	1.84	50.9	1.59
2048x2048x1024	64	91.0	1.42	83.0	1.30
2048x2048x2048	128	155.2	1.21	147.1	1.15
4096x2048x2048	256	283.4	1.11	275.4	1.08
4096x4096x2048	512	539.8	1.05	531.8	1.04
4096x4096x4096	1024			1044.3	32.63

16*: Spatial block size 1024x 512x 512

B. *Varing Blocking Optimizations for Fixed Problem*

To evaluate the effectiveness of the temporal blocking algorithm, we compare two types of executions for 7-point stencil computation for a 64-GiB problem, as shown in Fig. 18. *Type-1* represents the temporal blocking algorithm designed for 32 GiB memory. It employs two-level blocking, temporal and spatial blockings for cache and main memory hits. *Type-2* represents the best method and usual manner of full memory execution. It employs one-level space blocking for cache hits. *Type-2* takes 616 sec, which is fastest, when it runs on 128 GiB of memory; however, it takes 7,582 sec when it runs on 32 GiB of memory. *Type-1 with temporal blocking* takes 729 sec when running on 128GiB. This is because of temporal blocking overhead, redundant calculations and extra data space. *Type-1 without temporal blocking* performs only two-level space blocking. It takes 631 sec, which is slightly longer than the best time, 616 sec, because of redundant spatial blocking. When it runs on 32 GiB of memory, it takes 15,669 sec, because it ignores locality access for flash. Finally, *Type-1 with temporal blocking*, our proposed method, takes 1,163 sec on a 32-GiB memory. It only takes1.8 times longer than the best time, which is obtained by using *Type-2* on full memory.



type	block size	internal block size	temporal block size
1	1024-1024-512	32	128 or 1
2	2048-2048-1024	32	1

Fig. 18. Core times of 2 types of algorithms on various kernels

Fig. 20 and Fig. 21 show the swap device I/O bandwidth profiles when using the proposed algorithm (Type-1 with temporal blocking, blocking size (bt) is 8 time-steps) and the standard algorithm (Type-2 without temporal blocking, bt is 1) on 32 GiB of memory, respectively. They show the first 16 time steps of a 7-point stencil computation. The first part, where the write operation is dominant, is the data initialization part. In our algorithm, a higher read bandwidth is maintained at every sub-block calculation and the execution is completed sooner. Read corresponds to moving data from the domain buffer to the block buffer for temporal blocking. Write corresponds to writing the calculated data in the block buffer to the domain buffer. We can see that the same pattern is repeated 16 times in Fig. 19. One repeated pattern corresponds to one sub-block calculation. In this experiment, the domain is divided into eight sub-blocks, and one sub-block calculation includes 8-time-step updates of the data in sub-block buffer. The regions with zero bandwidth in Fig. 19 correspond to sub-block calculation parts. On the other hand, Type-2 in Fig. 20 shows a lower bandwidth profile and takes longer than Type-1 in Fig. 19. Its total amount of write data is 1.67 time larger than that of Type-1.

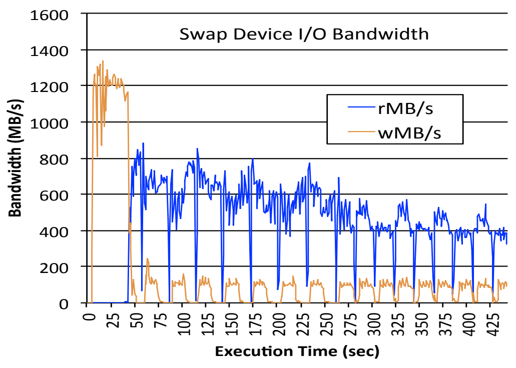


Fig. 19. Swap Device I/O bandwidth for stencil with 2-level temporal blocking (2048 x 2048 x 1024 double 64 GB+, 16 ite. one socket, 8 threads).

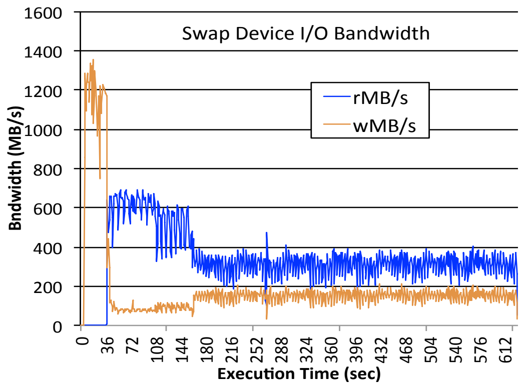


Fig. 20. Swap Device I/O bandwidth for stencil without temporal blocking, 2-level space blocking (2048 x 2048 x 1024 double 64 GB+, 16 ite. one socket, 8 threads).

C. *Varing Problem size for Fixed Physical Memory size*

Our next experiment measures performance by fixing the physical memory size to 32 GiB and varying the problem size from 16 GiB to 512 GiB, as shown in Table V. The actual data size is greater than the problem size, which corresponds to the total size of two spatial buffers with ghost regions for temporal blocking. Fig. 21 shows the relative performance using two sets of spatial and temporal block sizes: set-1 (bx, by, bz = 1024 x 1024 x 1024, bt = 96) and set-2 (bx, by, bz = 1024 x 1024 x 512, bt = 128). It shows how large a problem can be executed given a DRAM of limited size.

TABLE V. PROBLEM SIZE, ACTUAL DATA SIZE AND PHYSICAL MEMORY RATIO

	Spatial	1024x1024x1024		1024x1024x512	
	Temporal	96		128	
Problem Size	2 Buff Size (GiB)	Actual Size(GiB)	Ratio	Actual Size(GiB)	Ratio
1024x1024x1024	16*	25.1	1.57	27.3	1.71
2048x1024x1024	32	59.0	1.84	50.9	1.59
2048x2048x1024	64	91.0	1.42	83.0	1.30
2048x2048x2048	128	155.2	1.21	147.1	1.15
4096x2048x2048	256	283.4	1.11	275.4	1.08
4096x4096x2048	512	539.8	1.05	531.8	1.04
4096x4096x4096	1024			1044.3	32.63

16*: Spatial block size 1024x 512x 512

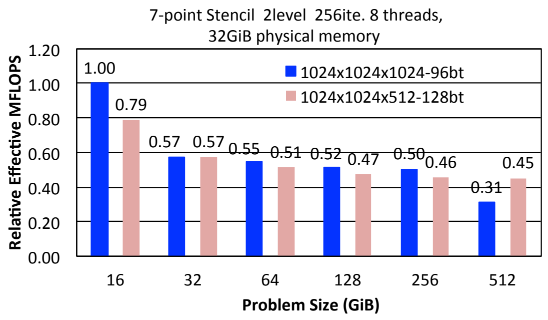


Fig. 21. 7-point stencil with 2-level blocking on 32 GiB memory (temporal block = 96 or 128, spatial 1st block = 1024 x 1024x 1024 or 1024 x 1024 x 512, 2nd block = 32 x 32 x 32, 256 ite. Physical mem. 32 GiB, 8 threads, fastswap).

According to Fig. 21, a 512 GiB-sized problem can run on a 32-GiB physical memory with an execution time nearly two times greater than the ordinary execution time on full memory. Fig. 21 shows that two sets of space block shape and temporal size influence the final performance. So an elaborated parameter tuning may gain better performance.

D. *Performance on New kernel 3.13.0 and NUMA server*

Our algorithm performs well on a two-socket NUMA server with 16 cores/2 CPUs under the fastswap. Moreover, it also works well under the newer kernel 3.13.0, in which the fastswap is now incorporated. Fig. 22 and Fig. 23 show the relative and absolute performance, respectively, when using a one-socket server under kernel 3.13.0 with 32 GiB of physical memory. It represents the performance of a 7-point stencil as well as 19-point and 27-point stencils, which include more calculations-per-point updates than a 7-point stencil. The relative performances for the calculations of a problem 16-times larger than the physical memory are 0.49 for 7-point, 0.65 for 19-point and 0.73 for 27-point. The performance depends on the ratio of calculation to the amount of memory access per iterative update. Fig. 24 and Fig. 25 show the case of 16 threads in a two-socket NUMA server listed in Table I. The relative performance is poorer than that shown in Fig. 22, but their absolute performance gains are better than those of one-socket server. In Fig. 24, 1 TiB problem of 27-point stencil

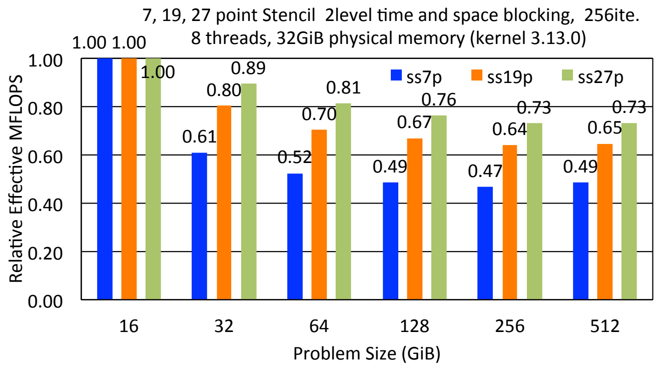


Fig. 22. Relative Effective Performance: 7-point, 19-point and 27-point stencil with 2-level blocking on 32 GiB memory (temporal block = 128, 1st spatial block = 1024 x 1024x 512 or 1024 x 512 x 512, 2nd spatial block = 32 x 32 x 32, 256 ite. 32 GiB, 8 threads, one-socket server).

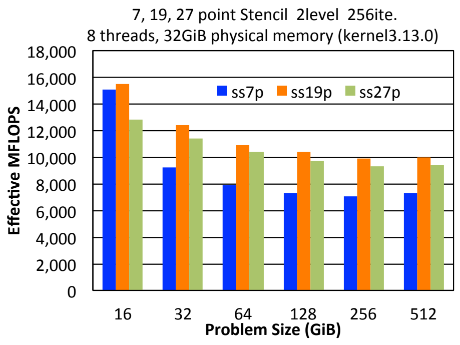


Fig. 23. Effective Performance: 7-point, 19-point and 27-point stencil with 2-level blocking on 32 GiB memory (temporal block = 128, 1st spatial block = 1024 x 1024x 512 or 1024 x 512 x 512, 2nd spatial block = 32 x 32 x 32, 256 ite. 32 GiB, 8 threads, one-socket server).

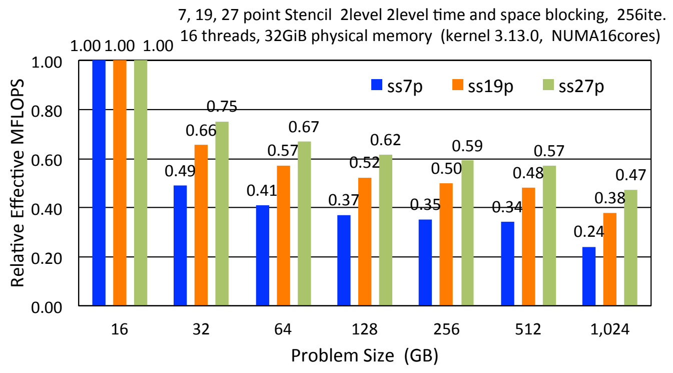


Fig. 24. Relative Effective Performance: 7-point, 19-point and 27-point stencil with 2-level blocking on 32 GiB memory (temporal block = 128, 1st spatial block = 1024 x 1024x 512 or 1024 x 512 x 512, 2nd spatial block = 32 x 32 x 32, 256 ite. 32 GiB, 16 threads, two-socket server)

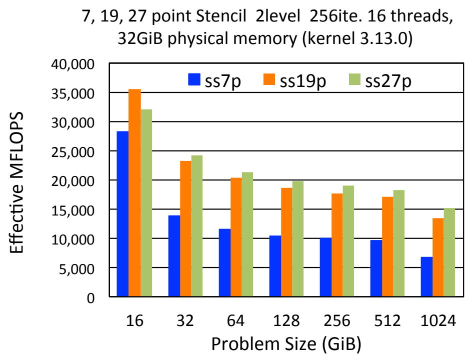


Fig. 25. Effective Performance: 7-point, 19-point and 27-point stencil with 2-level blocking on 32 GiB memory (temporal block = 128, spatial 1st spatial block = 1024 x 1024x 512 or 1024 x 512 x 512, 2nd spatial block = 32 x 32 x 32, 256 ite. 32 GiB, 16 threads, two-socket server).

VII. Related Works

Temporal blocking optimizations is not a new idea, but has mainly been applied thus far to cache and main memory tiers, the host memory and the graphics processing unit (GPU) memory [13][14][15] tiers, and to a local node and remote nodes in a cluster [12], in order to expedite data access by exploiting temporal locality. Some research in temporal blocking has been conducted in order to fine-tune performance depending on the cache, the non-uniform memory access (NUMA)-memory and multicore-CPU architectures [10][11][12]. Other studies have been conducted to generate general performance models and/or auto-tuning mechanisms independent of specific architectural parameters [16][17]. These studies aim to compensate for the latency gap between fast memories, like DRAM and cache. However, the gap between them is rather small compared to that between DRAM and flash. Thus, such techniques cannot be used directly for the tier between DRAM and flash. Our proposed algorithm in this preliminary evaluation is rather simple, but we believe it is the first attempt at applying temporal blocking optimization to the tier between DRAM and flash SSD, which is usually invoked through file I/O from applications.

Graph processing also employs memory-aware tuning from an application perspective in order to use a flash SSD as slow memory [18][19][20]. The breadth-first search (BFS) graph traversal found in Graph 500 [21] is a typical example. To process large amount of graph data, flash SSDs are used for the devices to partially offload the graph data. In contrast to stencil computations, graph processing has irregular memory access and low access locality. On the other hand, the data in graph processing can be categorized into a few groups, according to their attributes, such as access frequency and the manner of usage in each processing phase, dominated operation, read or write, the impact factor on the total performance, etc. The method employed here is highly tuned data partition and arrangement on flash and main memory for each processing phase, by using the knowledge of the data attributes, e.g. some parts of the graph data are rarely accessed. These methods are not adequate for general stencil computations, because all data are accessed uniformly and repeatedly and cannot be categorized to multiple types according to a different manner of use. However, more complex stencil computations that use multiple processing phases, and static and dynamic data arrangement using knowledge of data attributes will be effective.

In using flash SSD as memory, the I/O bottleneck caused by the Linux OS kernel is one of the biggest difficulties. It come from the overhead due to file systems, page caching, swap systems and memory and process space management. In stencil computation that uses flash as memory, there remains much room to accelerate performance by fine-tuning applications and by advances in OS kernel.

VIII. CONCLUSIONS

In this paper, we investigated the potential of flash SSD as large and slow memory for stencil computations, focusing on the case where a flash is used as swap device. Our study revealed that the fastswap patched kernel 3.6.0 yields better swap performance than kernel 2.6.32 of CentOS6.4 for stencil computations, but using a PCIe-based flash as a slow and large memory is not sufficient. We thus proposed a locality-aware, out-of-core computation algorithm using data structure blocking techniques on stencil computation to bridge the DRAM-flash latency divide. Our novel application of hierarchical temporal blocking optimization on stencil computation with a flash SSD performs satisfactorily for practical use. We find that 7-point and 27-point stencil computations for a 1-TiB problem (32 times that of the DRAM) using only a 32 GiB of DRAM and a flash SSD, in Mflops attain 24% and 47%, respectively, of the performance achieved in execution using only DRAM. There remains the possibility of achieving even better performance through more elaborate tuning, but our result nonetheless shows the potential and viability of flash SSD as large and slow memory.

We are currently evaluating the use of flash with a file map interface, not using it as a swap device. Our plan for future work includes automatic tuning for temporal and spatial blocking parameters for flash SSDs. We also intend to investigate the application of locality-aware algorithms to other high-performance computing applications. Our algorithm is not limited to the single tier between flash SSDs and DRAM, but extends to the hybrid implementation on other multiple memory tiers, e.g., GPU memory, host memory and flash SSDs.

REFERENCES

[1] Anirudh Badam, “How Persistent Memory Will Change Software Systems”, IEEE Computer, pp. 45-51, Aug. 2013.

[2] ioDrive2 (FusionIO) https://www.fusionio.com/products/iodrive2/.

[3] Intel SSD910 http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-910-series.html.

[4] Kshitij Sudan, Anirudh Badam and David Nellans, “NAND-Flash: Fast Storage or Slow Memory?”, NVM Workshop 2012.

[5] A. Badam and V.S. Pai, “SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy”, Proc. 8th Usenix Symp. Networked Systems Design and Implementation (NSDI11), Usenix, 2011,

https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Badam.pdf.

[6] Improve Linux swap for High speed Flash Storage http://events.linuxfoundation.org/sites/events/files/lcjpcojp13_shaohua.pdf.

[7] Nisha Talagala, “Creating Flash-Aware Applications”, Flash Memory Summit 2013, http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130814_203B_Talagala.pdf.

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

[9] STREAM benchmark, http://www.streambench.org/.

[10] Shaheen, M., Strzodka, R., “NUMA Aware Iterative Stencil Computations on Many-Core Systems ”, Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, DOI: 10.1109/IPDPS.2012.50, 2012 , Page(s): 461 - 473

[11] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann and Holger Fehske, “Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization”, Computer Software and Applications Conference, vol.1, pp. 579 – 586, 2009.

[12] 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, DOI: 10.1109/IPDPSW.2010.5470813

[13] Nguyen, A. ; Satish, N. ; Chhugani, J. ; Changkyu Kim ; Dubey, P. , “3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs”, High Performance Computing, Networking, Storage and Analysis (SC), 2010, DOI: 10.1109/SC.2010.2, 2010 , pp. 1 - 13

[14] Guanghao Jin, Toshio Endo and Satoshi Matsuoka, “A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU,” Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, pp. 1080 - 1087 , 2013, DOI: 10.1109/IPDPSW.2013.58

[15] 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, 10.1109/CLUSTER.2013.6702633

[16] K. Datta et al. “Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures”, Proc. SC2008. DOI:10.1145/1413370.1413375.

[17] M. Frigo and V. Strumpen, “Cache oblivious stencil computations”, Proc. ICS 2005. DOI:10.1145/1088149.1088197.

[18] Van Essen, B. ; Pearce, R. ; Ames, S. ; Gokhale, M., “On the Role of NVRAM in Data-intensive Architectures: An Evaluation”, Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, 2012, DOI:10.1109/IPDPS.2012.69

[19] Pearce, R. , Gokhale, M. , Amato, N.M. , “Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory”, IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 825 – 836, 2013, 2013, DOI: 10.1109/IPDPS.2013.72

[20] Keita Iwabuchi, Hitoshi Sato, Ryo Mizote, Yuichiro Yasui, Katsuki Fujisawa and Satoshi Matsuoka, “Hybrid BFS Approach Using Semi-External Memor”, International Workshop on High Performance Data Intensive Computing (HPDIC2014), in conjunction with IEEE IPDPS. May 2014 (to be published)

[21] Graph500, http://www.graph500.org