Blk-Tune: Blocking Parameter Auto-Tuning to Minimize Input-Output Traffic for Flash-based Out-of-Core Stencil Computations

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

***Abstract*—This paper proposes the auto-tuning system designed for flash-based out-of-core stencil computations. *Blk-Tune* is a runtime blocking parameter auto-tuning system that enables the use of flash memory as an extension of main memory. It incorporates automatic hardware information retrieval using Portable Hardware Locality and minimizes the amount of data transferred between the flash device and DRAM, which is the most dominant factor affecting the performance of out-of-core algorithms using flash. The use of explicit highly parallel asynchronous I/O to a flash device together with this auto-tuning system offers great advantages over the mmap method, in which a flash file is memory mapped. *Blk-Tune* allows users to easily achieve maximum performance of large-scale stencil computations in different hardware and application settings.**

Keywords- auto-tuning; non-volatile memory; flash memory; memory hierarchy; tiling; temporal blocking; stencil; out-of-core; asynchronous IO; mmap; block; memory extension

I. INTRODUCTION

Scientific computation often requires significant amounts of memory to attempt large-scale problems and/or for higher resolution data analysis. A common solution 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.

Today, flash memory has established its position in deep memory hierarchy as a cost-effective, power-efficient, and large-capacity type of memory behind the DRAM layer. Moreover, its performance has been improving [8 - 9], which makes it possible to use it as an extension of main memory for out-of-core algorithms [1, 6 - 7]. There are several ways in which a flash device can be used for memory extension, e.g., as a swap device, with file-mmap, and with explicit Input/Output (IO) operations [1 - 4]. Among these methods, the approach that has been attracting the most attention is the mmap method, where a flash SSD is used as a file system, and a file on the flash drive is mapped to the memory. Thus, mmap enables us to access a flash drive in the same manner as DRAM memory, in byte granularity, without any IO maintenance between the main memory and flash device. The most important advantage of mmap is that it requires little or no modification during application programming. The performance of applications using mmap depends on the degree of matching between the operating system (OS) kernel page replacement policy in the page cache and memory access patterns of the application. Unfortunately, an OS page replacement designed for general-purpose usage cannot be easily adapted for each application, even if the memory advice call, madvise(), is used. Hence, the performance improvement of applications is limited because of such implicit and general methods of page cache control in mmap. As a result, application-aware tuning is gaining more importance in performance–oriented fields.

On the other hand, the most explicit method of carrying out IO operations to a flash device, namely asynchronous IO (aio), requires application programs to be drastically restructured. That is, memory-semantic accesses to the flash drive in mmap are replaced with explicit IO operations. However, once the program is rewritten with an explicit aio, its performance is enhanced. The amount of data transferred between the flash device and memory can be efficiently controlled, unlike the redundant data transfer observed in OS-controlled page cache replacement in mmap. The aio method allows applications to maximally utilize the space on DRAM, because there is no need to reserve a page cache area as required by the mmap method. Moreover, users can design efficient IO scheduling to suit their applications. The use of aio with appropriate IO parameters allows IO data traffic to be minimized, in turn maximizing the application performance. A major problem with this approach is that explicit IO parameter tuning procedures are required when different system hardware settings and/or different application parameters are used. These hardware settings include the capacity of each memory layer and the number of CPU cores and sockets, and different application parameters such as the domain data size and number of time steps.

The first runtime auto parameter tuning system for flash-based out-of-core stencil computations was proposed in [3]. It uses a heuristic-based search and selects an acceptable parameter combination by employing ad-hoc procedures. In contrast, the new auto-tuning system proposed here is based on a global optimum search using a cost function to minimize the amount of IO traffic to the flash device. Stencil computations are one of the most important computational kernels in a wide variety of fields, such as scientific computing, engineering, and image processing. Auto-tuning for stencil computations has been studied by many researchers, who targeted GPU, general and many-core CPUs, and clusters [10-14], which are categorized into compiler-based to generate/

/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.48

transform codes, framework-based, and runtime-based methods. Our system, named *Blk-Tune*, provides runtime-based auto-tuning for using flash memory as an extension of the main memory. It incorporates runtime automatic hardware information retrieval using state-of-the-art Portable Hardware Locality [5]. It minimizes the amount of data transferred between the flash device and DRAM, which is the dominant factor affecting the performance of out-of-core algorithms using flash. The use of explicit IO operations to a flash device together with auto-tuning allows users to easily minimize the amount of IO traffic to achieve maximum performance for different hardware and application settings.

This paper first summarizes several algorithms and implementations of stencil computations, and compares their performance using both aio and mmap. Second, the newly developed auto-tuning system and procedures are introduced. Third, the effectiveness of *Blk-Tune* is evaluated. Finally, the conclusion and future work are presented.

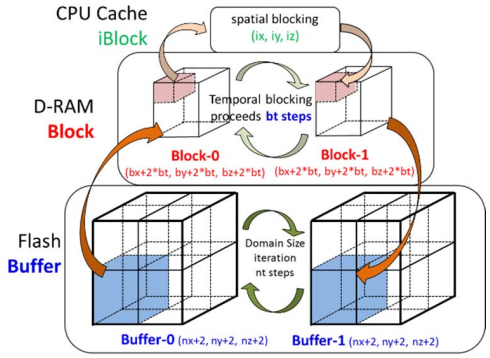

Figure 1. Three data arrays in memory layers, *Buffer arrays* in Flash SSD, *Block arrays* in DRAM, and *iBlock arrays* in the cache for locality extraction.

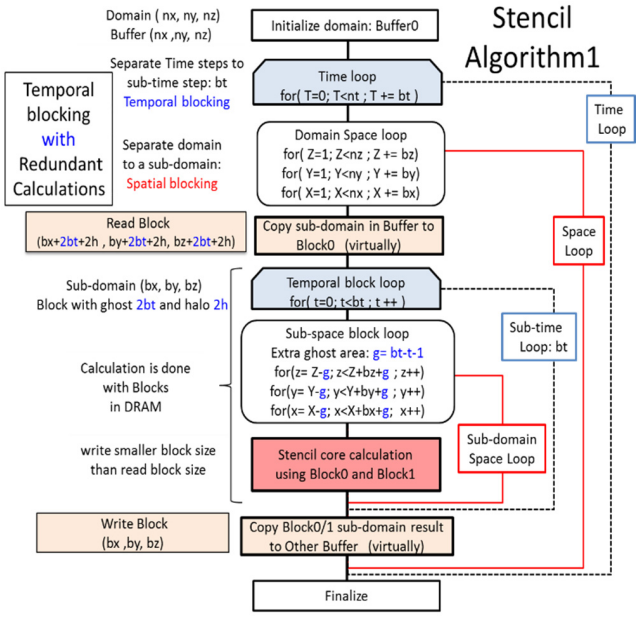

Figure 2. Temporal blocking algorithm-1 with redundant calculation: pseudo-codes for a 3D domain.

II. MMAP VS. AIO: IMPLICIT AND EXPLICIT METHODS IN TEMPORAL BLOCKING STENCIL ALGORITHM

A. Three methods for temporal blocking stencil algorithm

We have developed out-of-core stencil algorithms for flash SSDs using aio and mmap [1-3], by increasing data access locality using blocking techniques in spatial and temporal spaces as shown in Fig. 1. A typical temporal blocking algorithm, *algorithm-1*, for the three-dimensional (3D) data domain, is shown in Fig. 2 and Fig. 3. One block in source *Buffer array* is read to source *Block array* to calculate bt steps using two *Block arrays* in DRAM and the results in destination *Block array* are written back to destination *Buffer array* [1].

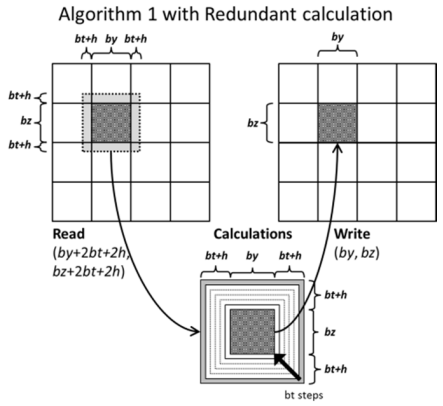

Figure 3. Calculation of bt steps in block arrays in DRAM and read/write from/to domain arrays in the flash device in *algorithm-1*.

B. Basic performance of three methods in algorithm-1

We have investigated several data layouts in memory and work-share schemes for the stencil computation for aio and mmap, as shown in Fig. 4 [3]. With these results, Fig. 5 compares the relative execution times of 7-point stencil computation using three methods, 1) using a flash as a swap device, 2) using file memory mapping (mmap), and 3) using asynchronous read/write to flash (aio). The platforms used in this paper are listed in Table I.

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. 5. The aio method with temporal blocking is most effective for execution under limited DRAM.

TABLE I. EXPERIMENTAL ENVIRONMENT

server	L1 cache (KiB)	L2 cache (KiB)	L3 cache (MiB)	Phys Mem (GiB)	Flash Mem (TiB)	CPU Xeon E5, (GiHz)	Total cores	socket	cores/ socket
crest0	32	256	20	32	1.2	2650, (2)	8	1	8
crest4	32	256	20	64	0.785	2687W, (3.1)	16	2	8
crest6	32	256	25	128	1	2687W v3,(3.1)	20	2	10

Fig. 6 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. 6).

C. *Performance of algorithm-1 using aio and mmap in NUMA- system*

Fig. 7 and Fig. 8 compare 7-point and 19-point stencil computation times and effective MFlops of the aio and mmap methods for problems of various sizes, i.e., ranging from 64 GiB to 1 TiB, in a non-uniform-memory architecture (NUMA) system. The execution time of the aio method is 50–60% of that of the mmap method. Moreover, during execution using the mmap method, the 19-point 1 TiB problem is terminated by an out-of-memory killer in the OS, because of the lack of memory availability in the large-size file mmap. In contrast, execution of the aio method exhibits stable behavior. Fig. 9 compares the aio and mmap methods in terms of MFlops for problems of various sizes on a system with a fixed DRAM size. The figure shows the problem size in terms of the *buffer array* size for the flash device and *block array* size for the DRAM. As the problem size increases, the DRAM capacity available for the page cache area decreases, thereby causing a larger performance difference between aio and mmap.

These results show the advantage of the aio method compared to the mmap method.

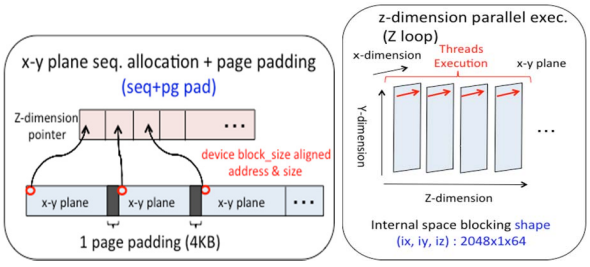

Figure 4. *Block array* memory layout for block-aligned access of the aio method (left), and work-share among threads in an *iBlock array* (right).

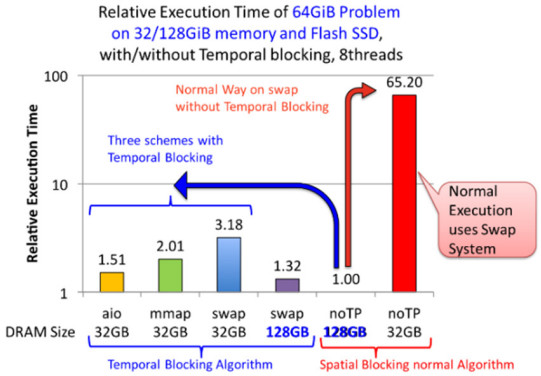

Figure 5. Relative times for various methods for 7-point stencil computation of 64 GiB-size problem using 32 GiB DRAM and a flash on single-socket system (crest0).

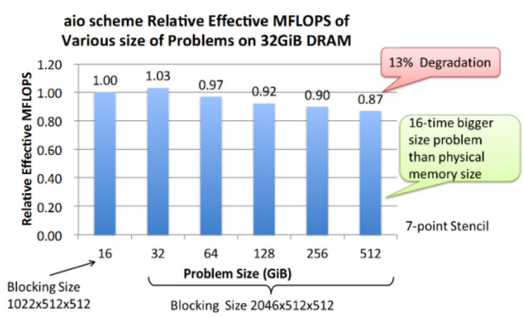

Figure 6. Performance for various-size problems (7-point stencil) on fixed physical memory (32GiB) in single-socket system (crest0)..

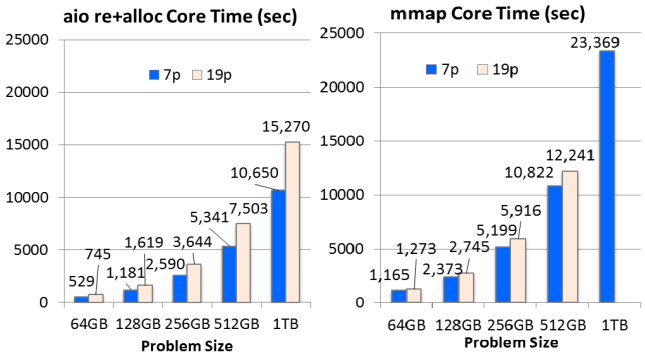

Figure 7. Execution times required by the aio and mmap methods for problems of various sizes on two-socket system (crest4).

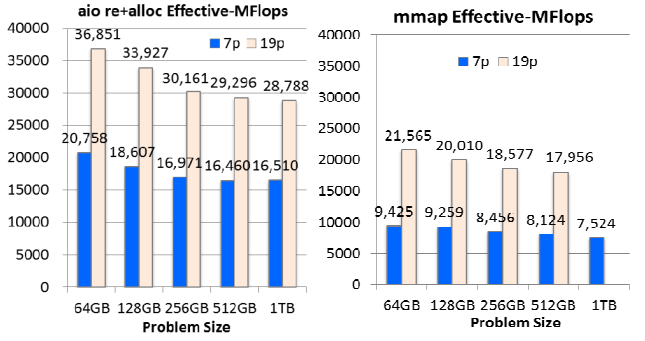

Figure 8. Effective MFlops of the aio and mmap methods for problems of various size on two-socket system (crest4).

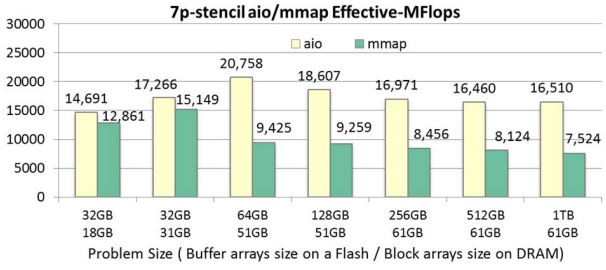

Figure 9. Effective MFlops on fixed-size memory (64 GiB) on two--socket system (crest4) for aio and mmap methods.

III. VARIOUS ALGORITHMS AND IMPLEMENTATIONS FOR FLASH-BASED OUT-OF-CORE STENCIL COMPUTATIONS

A. *Algorithm-2: temporal blocking without redundant calculation*

This paper introduces a new algorithm, *algorithm-2*, which performs temporal blocking without redundant calculations. The algorithm is based on sliding a spatial-block calculation window in block arrays on DRAM in temporal block time (*bt*) steps as shown in Fig. 10 and Fig. 11. In *algorithm-2*, the size of the read and write blocks are the same in contrast to *algorithm*-1, in which the read and write block sizes differ as shown in Fig. 3.

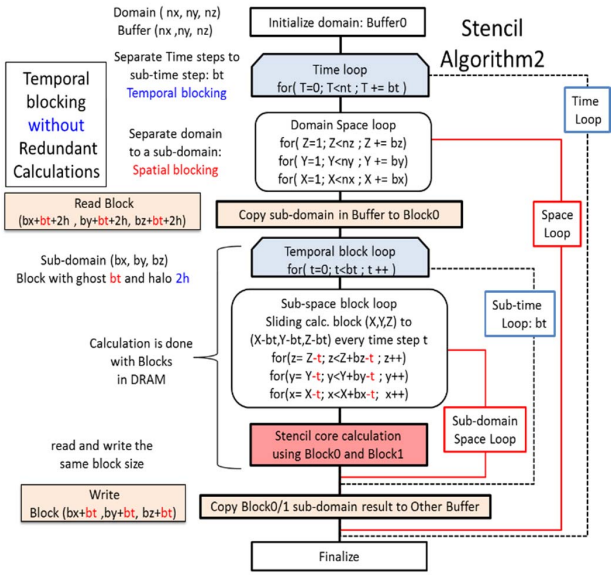

Figure 10. Temporal blocking *algorithm-2* without redundant calculation: pseudo-codes for a 3D domain.

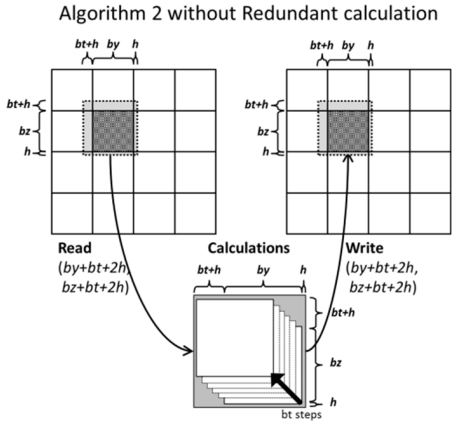

Figure 11. Calculation of bt steps in block arrays in DRAM and read/write from/to domain arrays in the flash device in *algorithm-2*.

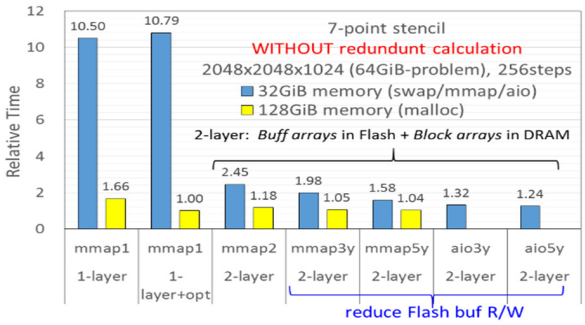

Figure 12. Relative execution times of various implementations with the aio and mmap methods in *algorithm-2*

B. *Performance of various algorithms using mmap and aio.*

Fig. 12 shows the relative execution times of several versions of algorithm-2 using mmap and aio. The program *mmap1*, i.e., the original version with no redundant calculations, which is designed for main memory and cache layers without a flash device, does not have an intermediate layer, the *Block arrays* shown in Fig. 1. In *mmap1*, cache-DRAM maintenance is implicitly performed by hardware. When employing the *mmap1* program for a flash-based out-of-core computation using the file-mmap method, its DRAM-Flash maintenance is implicitly carried out as page cache maintenance by the OS kernel in mmap. Fig. 12 compares *(1) mmap1+opt*, which is a modified version of *mmap1* by applying our optimization [3] shown in Fig. 4; *(2) mmap2*, which introduces an intermediate layer, *Block arrays* on DRAM; and two versions of advanced algorithms *(3) mmap3y* (or *aio3y*) and *(4) mmap5y* (or *aio5y*), which minimize the total amount of data transferred to the flash device. The original algorithms, *mmap1* and *mmap1+opt*, are fast, when a sufficient amount of memory (128 GiB) is available for use. However, their performance deteriorates when using only 32-GiB DRAM, which is half the size of the problem (64 GiB). This situation is attributed to implicit page caching in mmap that causes inefficient data transfer to the flash device, as a result of using only *buffer arrays*, without an intermediate *Block array* layer. In comparison, two-layer algorithms using intermediate *Block arrays* in DRAM improve their performance considerably when 32 GiB DRAM is used.

When using a large temporal blocking size, bt, in *algorithm-1,* there is a tradeoff between performance acceleration by increasing temporal access locality and performance degradation by increasing redundant calculations. On the other hand, as the *algorithm-2* is free from the redundant calculation overhead, there is only the tradeoff between spatial and temporal blocking sizes that share DRAM with fixed-capacity. Evaluation of the blocking parameter combinations with the amount of IO traffic they generate under the fixed-capacity of DRAM is an effective way to find the best blocking parameter combination in *algorithm-2.* Among the various implementations of *algorithm-2*, aio3y and aio5y, which use the aio method, are the fastest; hence, they are employed for the evaluation of *Blk-Tune*. Details of these algorithms are omitted here because of the space limitation.

IV. AUTO-TUNING SYSTEM FOR FLASH-BASED OUT-OF-CORE STENCIL COMPUTATIONS

The retrieval of hardware information regarding the underlying platform during runtime and the use of the information to estimate the total amount of IO traffic to a flash device, enables the selection of an optimal combination of spatial and temporal blocking sizes to suit the capacity of each memory layer (flash, DRAM, level-3 cache, and level-2 cache). This auto-tuning mechanism allows users to easily minimize the amount of IO traffic and gain maximum performance for particular hardware and application settings.

The auto-tuning system, named *Blk-Tune*, for flash-based out-of-core stencil computations, is newly proposed. Fig. 13 shows an overview of the system. The system obtains platform hardware information in runtime by using Portable Hardware Locality (hwloc) [5], which is widely portable to various kinds of OS and CPU architectures. *Blk-Tune* is available for online tuning as a frontend of stencil computations in runtime, as well as for offline analysis when hardware information is input manually. The offline tuning capability enables us to tune parameters in any environment that differs from the platform used to run applications. Once the problem size and platform are fixed, *Blk-Tune* produces an optimal blocking parameter combination to minimize IO traffic to flash. In online tuning, 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.

CalcTrans in Fig. 13 calculates the total amount of IO traffic for a specified block size (*bx, by, bz, bt*). This calculation depends on stencil algorithms, as shown in Fig. 3 and Fig. 11. Thus, multiple CalcTrans functions are prepared and an appropriate one is available for the stencil algorithm used in the backend stencil computations. In the current implementation, the parameter search for *Block* and *iBlock* in Fig. 13 is based on a global optimal search to minimize the cost function under several constraints defined in section V.

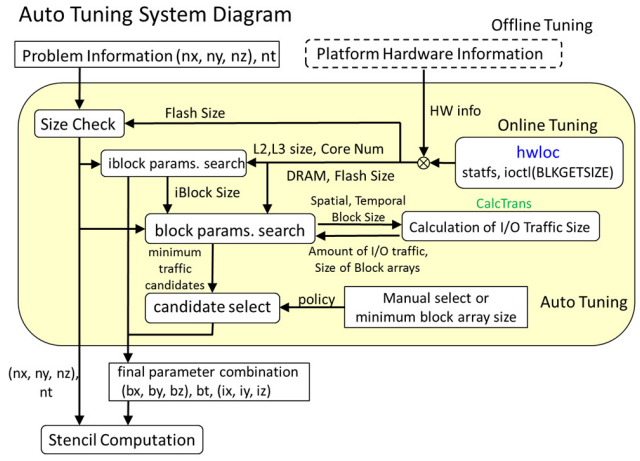

Figure 13. Auto-tuning system for out-of-core stencil computations using flash, DRAM, and cache. It is available both for online tuning in application runtime and for offline tuning for the analysis of algorithms.

The following defines the terminology and problem definition in this paper.

A. Inputs to the auto-tuning system

1) Hardware information

It is automatically supplied by hwloc [5] in runtime or input by users for offline analysis.

a) Capacity of each memory layer

Sc2: level-2 cache capacity, for each CPU core
Sc3: level-3 cache capacity, shared among CPU cores
Sm: Main memory (DRAM) capacity,
Sf: Flash memory capacity
Sb: Flash device block size

b) CPU configurations

Nc: # of cores in a system
Ns: # of sockets

2) Problem information

It is input by users.

a) Domain data size

nx, ny, nz: size in each dimension in 3D array

b) Number of iterations for stencil computations

nt: (time steps)

B. Outputs from the auto-tuning system

1) An optimal combination of temporal and spatial block size

bx, by, bz: size in each dimension in 3D array
bt: temporal size in time dimension

The output combination causes minimum IO traffic between DRAM and flash memory, and the volume of the *Block array* generated by this combination can be accommodated in main memory (DRAM) capacity.

2) Optimal internal spatial block size combination ix, iy, iz : size in each dimension in the 3D-array

Internal spatial block, *iBlock* (*ix, iy, iz*), is processed in parallel by multiple threads and the output internal block size combination maximally utilizes the aggregation of level-2 caches in CPU cores in a system.

C. Premises

The problem parameters input by the user are required to satisfy the following conditions.

1) 3D domain data arrays have to fit the specified flash device.

$$Sf > nx * ny * nz * k * Esz$$

k is constant, typically 2 for double buffer arrays.
Esz is the data element size in byte.

2) nx is required to be a multiple of Sb.

This premise is necessary to achieve efficient parallel asynchronous IO by multiple threads, in which flash-block-size- aligned data layout in memory is required.

V. AUTO-TUNING PROCEDURES OF BLOCKING PARAMETERS FOR STENCIL COMPUTATIONS

The tuning procedure consists of three steps: 1) optimize internal block parameter *(ix, iy, iz),* 2) optimize block parameter *(bx, by, bz, bt),* and 3) select the final parameter combination.

A. Step 1: find optimal iBlock size (ix, iy, iz)

Find the optimal combination of internal block spatial size, *ix, iy,* and *iz* that maximally utilizes the aggregation of level-2 caches in CPU cores in a system. In the current implementation, it is assumed that each level-2 cache exists in one core, and the level-3 cache is shared by cores in a CPU socket. If the volume of *iBlock* is larger than the total capacity of level-2 caches under the constraints below, the output combination selects the one which has the nearest volume of the total size of level-2 cache in cores.

1) Objective Function

$$min\,(\,|\;k * iBsz(\,ix, iy, iz) * Esz - Sc2 * Nc\,|\,) \quad (1a)$$
$$where\;\; iBsz(ix, iy, iz) = ix * iy * iz.$$

k is a constant, typically 2, for double buffer arrays. *Esz* is the data element size in bytes.

2) Constraint conditions

a) iz must be a multiple of Nc.

This requirement is one of the heuristics. It is preferable to ensure a balanced workload distribution among multiple threads to achieve efficient work sharing and reduces synchronization overhead.

b) ix has to equal nx.

In our flash-based algorithm, this condition is required to ensure asynchronous IO data alignment for efficient asynchronous parallel IO by multiple threads. Fig. 14 shows the relationship between a block array *(bx, by, bz)*, and an internal block *(ix. iy. iz)*.

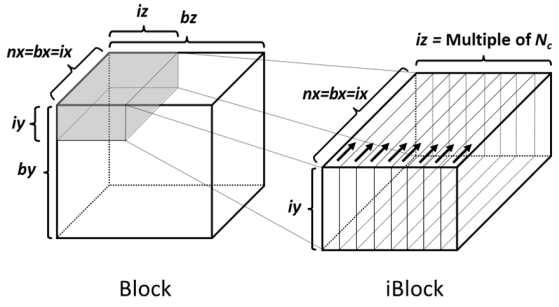

Figure 14. Finding the optimal *iBlock* size, the spatial blocking size of a *Block* array, for efficient multi-thread calculations.

B. Step 2: find the optimal Block size (bx, by, bz) and bt

Find an optimal combination of block spatial size, (bx, by, bz), and temporal size *bt,* which minimizes IO traffic between DRAM and Flash memory in stencil computation. Generally, larger spatial and temporal sizes that do not exceed the available DRAM capacity derive higher performance. A tradeoff among the spatial and temporal sizes, *bt, bx, by,* and *bz,* exists.

1) Objective function

$$\min(Total\_A) = \min\left(\sum_{i}^{T_c} Ai\right)$$
$$= \min\left(\sum_{i}^{T_c}\sum_{X,Y,Z}^{nx,ny,nz} B(X,Y,Z,i)\right) \quad (2a)$$

Total_A is the total amount of IO traffic (in bytes) during the entire stencil processing procedure of *nt* steps. The objective function minimizes the total IO traffic to a flash device. T_c is the total number of sweep iterations that sweeps over the entire domain data array, i.e., the *Buffer array* in Fig. 1. It corresponds to the outer loop of time step T in Fig. 2 and Fig. 10.

$$T_c = ceiling\left(\frac{nt}{bt}\right), \quad T_f = floor\left(\frac{nt}{bt}\right) \quad (2b)$$

Ai is the amount of IO traffic in the *i'th* sweep iteration,

$$Ai = \sum_{X,Y,Z}^{nx,ny,nz} B(X,Y,Z,i) \quad (2c)$$
$$where\;\; 1 \le i \le T_c,$$
$$1 \le X \le ny \;\; and \;\; X \; is \; a \; multiple \; of \; bx,$$
$$1 \le Y \le ny \;\; and \;\; Y \; is \; a \; multiple \; of \; by,$$
$$1 \le Z \le nz \; and \; Z \; is \; a \; multiple \; of \; bz.$$

B(X, Y, Z, i) is the amount of IO traffic for calculating a block of which the starting point coordinates are *(X, Y, Z)* in the *i'th* sweep iteration. The summation in (2c) corresponds to the outer-space loop, *X, Y,* and *Z* in Fig. 2 and Fig. 10.

According to the relationship between the number of sweep iterations, T_c, T_f, and nt in Fig. 15, the following (2d) can be derived.

$$\sum_{i}^{T_c} Ai$$
$$= \sum_{X,Y,Z}^{nx,ny,nz} \left(B(X,Y,Z,1) * T_f + B(X,Y,Z,T_c) * \left(T_c - T_f \right) \right) \quad (2d)$$

The first term represents the IO traffic in the majority of cases of the sweep iteration, which can be represented as the IO traffic in the first iteration multiplied by T_f. The second term in *(2d)* corresponds to the IO traffic in the last sweep iteration,

T_c, and it is negligible when *Tc* equals T_f. The last iteration uses a smaller temporal blocking size, *nt* modulo *bt.*

$$bt_i = bt, \quad where \;\; 1 \le i \le T_f$$
$$bt_i = nt \text{ modulo } bt, \quad \text{where} \;\; i = T_c, \;\; T_c \neq T_f.$$

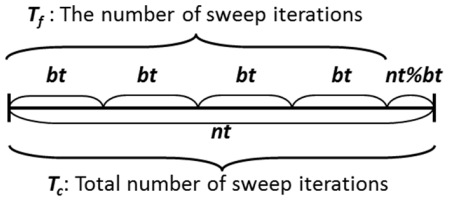

Figure 15. Relationship between T_c and T_f.

The amount of IO traffic in each block, *B(X, Y, Z, i),* is different, depending on the stencil algorithms and their position *(X, Y, Z)* in a block division scheme. Details of *B(X, Y, Z, i)* are provided in section C.

2) Constraint conditions

a) bz must be multiple of iz

b) by must be multiple of iy

Both conditions are preferable for a balanced spatial blocking of one *Block* array as shown in Fig. 14. It causes a balanced workload distribution among multiple threads in each *y* and *z* dimension.

c) bx must equal nx, bx= nx

This condition is required when using an aio-based algorithm and is important to achieve highly efficient asynchronous parallel IO from/to the flash device. It means *bx* is a multiple of the flash device block size, *Sb,* and that access to any point of (*y, z*) in Buffer arrays and Block arrays is *Sb*-block-aligned. It guarantees that individual parallel IO issued by multiple threads becomes a large-sequential-size read and write operation aligned in *Sb*. In our algorithm for aio, a 3D domain buffer array is treated as a 2D buffer array of Y and Z, with an X-element column shown in Fig. 14. It can be treated as a 2D (*by* × *bz)* array with an *nx*-element, as shown in Fig. 3 and Fig. 11.

d) bt must satisfy the following,

$2 \le bt \le nt, \;\; bt \le \frac{bx}{2}, \;\; bt \le \frac{by}{2}, \;\; and \;\; bt \le \frac{bz}{2}$

bt = 2: temporal blocking for minimum size

The temporal blocking size, *bt,* must not exceed half of the spatial blocking sizes *bx, by,* and *bz* in each dimension. It must be larger than 1 and should not be larger than *nt*. If *bt* = 1, it means the algorithm does not use temporal blocking.

e) Total size of block arrays must not exceed DRAM capacity

$$k * Bsz\_max * Esz < Sm - Km$$

Bsz_max is the size of the largest block among the blocks used in the entire stencil computation. *k* is constant, typically 2, for double block arrays. The size of two block arrays in DRAM must fit the available DRAM capacity, *Sm*. *Km* is the constant memory capacity reserved for the OS.

Bsz_max is the number of data elements in the maximum-size block among blocks at *(X,Y,Z)*. *Bsz_r(X,Y,Z,1)* and *Bsz_w(X,Y,Z,1)* are the read and write size of the block array at *(X,Y,Z)* in the first sweep iteration.

$$Bsz\_max = Max_{X,Y,Z}\big(Max\,(Bsz_{w(X,Y,Z,1)}, Bsz_{r(X,Y,Z,1)})\big)$$

$$where \;\; X = 1, 1 \le Y \le ny \;\; and \;\; Y \; is \; a \; multiple \; of \; by,$$
$$1 \le Z \le nz, and \; Z \; is \; a \; multiple \; of \; bz.$$

C. Amount of IO traffic for a Block: B(X, Y, Z, i)

B(X, Y, Z, i) used in the objective function *(2a)* is the amount of IO traffic for calculating a block at (*X, Y, Z*) in the *i-th* sweep iteration. Typically, it is expressed as follows.

$$B(X,Y,Z,i) = \big(kr * Bsz_{r(X,Y,Z,i)} + kw * Bsz_{w(X,Y,Z,i)}\big) * Esz,$$
$$where \;\; 1 \le kr \le 2, \;\; 1 \le kw \le 2 \qquad (2e)$$

where *Esz* is the data element size in bytes, *kr* and *kw* are constant, depending on the stencil algorithms, e.g., *kr* = *kw* = 2 when source and destination buffers are read and write. *kr* = *kw* = 1 when source buffer read and destination buffer write are required. Typical read and write block size, Bsz_r*(X, Y, Z, i) and* Bsz_w*(X, Y, Z, i)* are expressed with temporal block size, *bt*, and ghost (or *halo*) size, *h,* the number of neighboring elements in each direction in each dimension for a one-element update in a stencil calculation.

1) Typical block array size for algorithm-1

The typical block array size for *algorithm-1* with redundant calculation in Fig. 3 is as follows.

$$Bsz1\_r = (bx + 2 * h + 2 * bt) * (by + 2 * h + 2 * bt) * (bz + 2 * h + 2 * bt)$$

$Bsz1\_r$: read block size from source (src) buffer in flash memory to src block in DRAM

$$Bsz1\_w = (bx) * (by) * (bz)$$

$Bsz1\_w$: write block size to destination (dest) buffer in flash memory from dest block in DRAM

Total amount of IO traffic required for one block update in *algorithm-1* is as follows.

$$B1(X,Y,Z,1)typical = (Bsz1\_r + Bsz1\_w) * Esz \;\; (2f)$$

2) Typical block array size for algorithm-2

The typical block array size for *algorithm-2* without redundant calculation in Fig. 11 is as follows.

$$Bsz2\_r = Bsz2\_w = (bx + 2 * h + bt) * (by + 2 * h + bt) * (bz + 2 * h + bt)$$

$Bsz2\_r$: read block size from src/dest buffers in flash memory to src/dest blocks in DRAM

$Bsz2\_w$: write block size to src/dest buffers in flash memory from src/dest blocks in DRAM

Total amount of IO traffic required for one block update in *algorithm-2* is as follows.

$$B2(X,Y,Z,1)typical = 2 * (Bsz2\_r + Bsz2\_w) = 4 * Bsz2\_rw * Esz \quad (2g)$$

3) Typical block array size when halo equals one

In this paper, the ghost element size *h* is assumed to be one for simplicity. It corresponds to typical cases, such as 7-point, 19-point, or 27-point stencil computation. Moreover, *bx* equals *nx*, according to the constraints. Thus, *bt* can be omitted for the *x*-dimension. As a result, typical IO traffic of block update in *algorithm-1* and *algorithm-2* are as follows.

$$B1(X,Y,Z,1)typical = ((nx + 2) * (by + 2 + 2 * bt) * (bz + 2 + 2 * bt) + (nx + 2) * (by + 2) * (bz + 2)) * Esz \quad (2f')$$

$$B2(X,Y,Z,1)typical = 4 * (nx + 2) * (by + 2 + bt) * (bz + 2 + bt) * Esz \quad (2g')$$

4) Division scheme and boundary effects in block size

The actual read/write block array sizes are different from each other depending on the block position *(X, Y, Z)* in the domain data array and a division scheme. Fig. 16 - Fig. 18 show variations in the IO size of a block array for *algorithm-2* influenced by division schemes and domain data boundary conditions. Thus, the use of the typical block sizes in equations *2f* and *2g* is not always adequate for the precise evaluation of the amount of IO traffic and block size. In Fig. 16, all block sizes estimated with the typical block size in the box on the right are larger than the actual maximum block size, *Bsz_max*, on the left. Larger *bt* values usually derive higher performance for flash-based out-of-core stencil computations; thus, precise evaluation derives improved spatial and temporal optimization for DRAM capacity.

Fig. 17 and Fig. 18 show two cases to illustrate the relationship between the required maximum block size, *Bsz_max,* and division schemes. In Fig. 17, the balanced division scheme in the lower part of the figure is more effective than the unbalanced scheme in the upper part because a smaller block size favors spatial optimization, as well as IO traffic optimization. On the other hand, Fig. 18 shows that the unbalanced division scheme in the upper part of the figure requires smaller blocks compared to ones in the balanced division shown in the lower part. In this case, the balanced division sets *Bsz_max* equal to the total size of *by'* of Block-B plus *bt+h*, which exceeds the size of *by* of Block-A in the upper scheme. Thus, the unbalanced division is more efficient for spatial optimization.

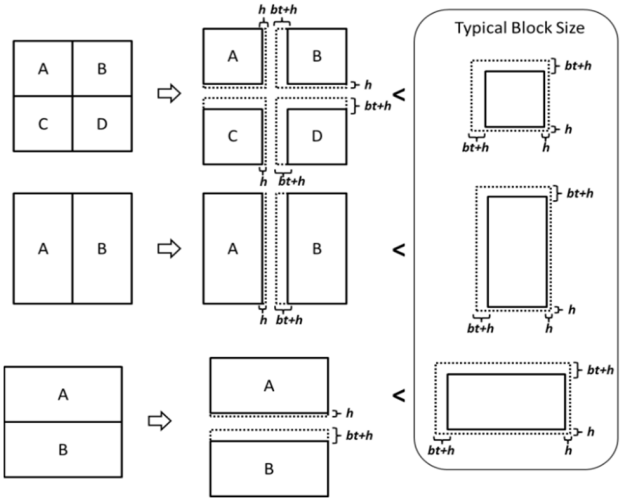

Figure 16. Variations of the IO size of a block array influenced by dividing schemes and domain data boundary conditions. The use of the typical block sizes, shown on the right, is not adequate for precise evaluation of the amount of IO traffic. Precise block size evaluation also derives improved spatial optimization for DRAM capacity.

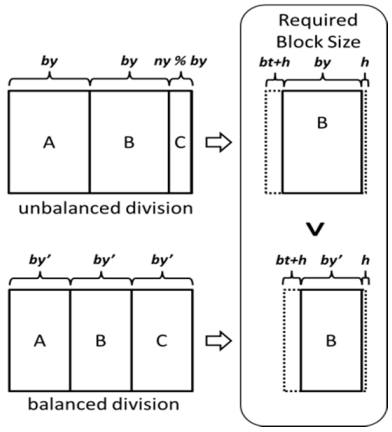

Figure 17. Difference in the IO size of a block array influenced by the relationship between bt size and division schemes. In this case, a balanced division scheme is more effective than an unbalanced scheme. The smaller IO size of the division shown below leads to improved performance.

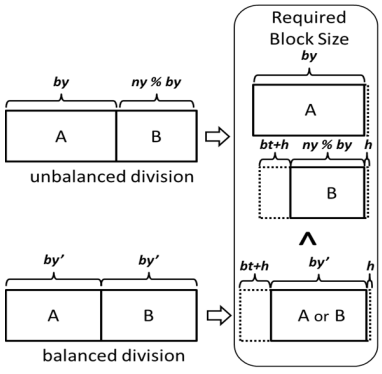

Figure 18. Difference of IO size of a block array influenced by relationship between *bt* size and division schemes. In this case, the performance of an unbalanced division scheme exceeds that of the balanced scheme.

D. Step 3: Select the final parameter combination

If there are multiple candidates of combination *(bx, by, bz, bt)* that have the same minimum amount of IO traffic, the current tuning implementation selects one combination that uses the smallest size of the block array, *Bsz_max*, which means it leaves more room in DRAM for other OS activities.

VI. Evaluation of *Blk-Tune* system

The fastest implementations, aio3y and aio5y using *algorithm-2* in Fig. 12, are used for the evaluation of *Blk-Tune*. In this section, *Blk-Tune,* the first version of the auto-tuning system named *Naïve-Tune* used in [3], and *manual parameter selection* are compared.

Manual parameter selection entails choosing arbitrary acceptable blocking parameters manually without any calculation by computers. It first selects a largest power of two number combination (*by and bz)* as spatial blocking size. The volume of *Block arrays* (*bx*by*bz *2)* is required to be smaller than the given DRAM capacity. Memory capacity is usually a number that is a multiple of power of two; thus, it is not so difficult for persons to choose an acceptable largest combination, *bx, by, and bz.* This manual procedure is simulated in the *Naïve-Tune* system. After spatial blocking parameters are determined, a temporal blocking size *bt* is chosen among divisors of *nt* steps, usually a modest value, because the typical *Block array* volume in equation *2g'* is required not to exceed DRAM capacity without precise calculation.

A. Naïve-Tune system

The Naïve-Tune system does not search for a global optimal blocking size combination to maximize the volume of spatial and temporal size that can be accommodated by the available DRAM capacity, unlike the *Blk-Tune* system. It does not use any cost function, e.g. the amount of IO traffic used in *Blk-Tune*. *Naïve-Tune* automatically retrieves platform information and selects an acceptable combination of spatial dimension size, *by and bz,* by using heuristic-based procedures. In *Naïve-Tune,* the spatial blocking sizes, *by* and *bz,* are required to be power of two numbers and they are required to be the same values *(by = bz)* or a doubled value of the other (*by = 2*bz*, or *bz = 2*by*). There are more restrictions in search space compared to those in *Blk-Tune*. Moreover, the estimation of *Block array* volume in *Naïve-tuning* is based on only typical cases described in equation *2f* and *2g*. After the spatial block sizes, *bx, by*, and *bz*, are determined, *Naïve-tuning* selects a temporal blocking size, *bt*. Thus, there is no evaluation of the trade-off between spatial and temporal sizes under fixed-size of DRAM. On the other hand, *Blk-Tune* evaluates all possible combinations of temporal and spatial combinations. That is, the *Naïve-tune* system has a limited ability compared to *Blk-Tune*, although it finds an appropriate blocking parameter combination fast in runtime. Thus, the *Naïve-tune* system is helpful compared to using a trial-and-error manual parameter selection.

B. Tuning procedures in Naïve-Tune system

Step 1. Search maximum volume of spatial block sizes, (*bx, by, bz)* that can be accommodated in the DRAM of *Sm* size. The constraint conditions used here are 1) *bx* must equal *nx*, 2) *by* and *bz* must be a power of two, 3) *by must equal bz,* or *by must equal 2*bz*, or *bz equals 2*by*.

Step 2. Search for the maximum temporal blocking size *bt_max* could be accommodated in the DRAM when using the block sizes (*bx, by, bz)* selected in *Step 1*. Only the typical block size calculation in *(2f')* is used for the estimation of the volume of the block size. There is no mechanism to take account of the effects of division schemes and the domain array boundary conditions shown in Fig.16, 17 and 18.

Step 3. Update *bt_max* to *bt*, to ensure a balanced division size in the time dimension as shown in Fig. 15. In other words, find *bt* such that the distance of *nt modulo bt* is minimized while retaining the same number of sweep iterations, T_c.

$$min(\,|\,(nt\ modulo\ bt) - bt\,|\,),$$
$$where\ \ ceiling\left(\frac{nt}{bt}\right) = ceiling\left(\frac{nt}{bt\_max}\right) = T_c$$

This heuristic is also incorporated in *Blk-Tune* search.

C. Performance of Blk-Tune vs. Naïve-Tune

Fig. 19 compares the results obtained for *Blk-Tune, Naïve-Tune,* and *manual parameter selection* in stencil execution time for the 128 GiB problem (*nx, ny, nz, nt* = 2048, 2048, 2048, 256) on crest4 server (16 cores, 32 GiB DRAM). It also shows parameter combinations for *iy, iz, by, bz,* and *bt,* used here. According to the constraint conditions described in section IV, all blocking sizes in *x*-dimension equal *nx* (*bx= ix=nx).*

The times obtained by *Blk-Tune* in Fig. 19 correspond to 68%-73% of the times obtained for *manual parameter selection*, and 75%-80% of the times obtained by *Naïve-Tune* for both aio5y and aio3y. Fig. 20 shows the time components for each execution in Fig. 19: flash-read *(buf2blk)*, flash-write *(blk2buf)* and memory copy *(blk2blk),* and stencil calculations. Fig. 21 shows the actual amount of IO traffic for read/write operations from/to the source/destination in each tuning system when using the *iBlock* parameter (*iy, iz* =1,128).

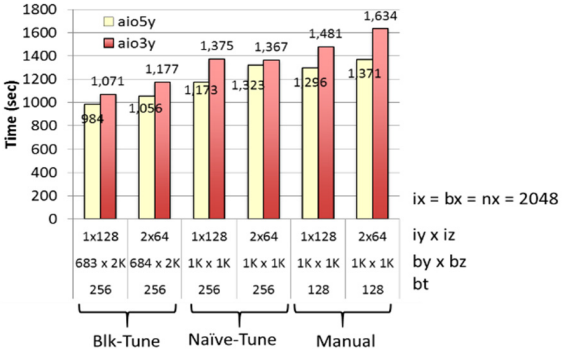

Figure 19. Selected parameter settings and execution times for the 128 GiB problem obtained by three tuning systems for the two algorithms, aio5y and aio3y on crest4 system.

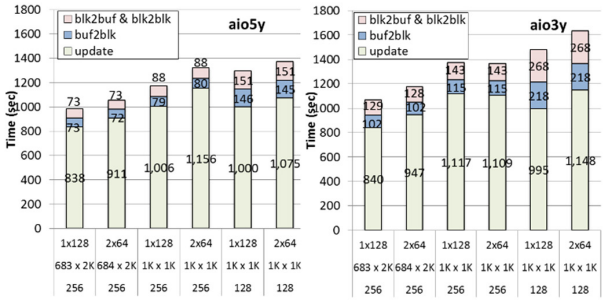

Figure 20. Time components for each of the tuning systems for aio5y and aio3y on crest4 system.

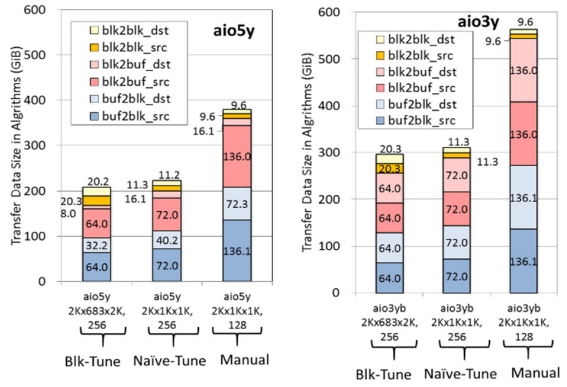

Figure 21. Actual amount of IO traffic in *Blk-Tune, Naïve-tune*, and *manual selection* for aio5y (left figure) and aio3y (right figure) on crest4.

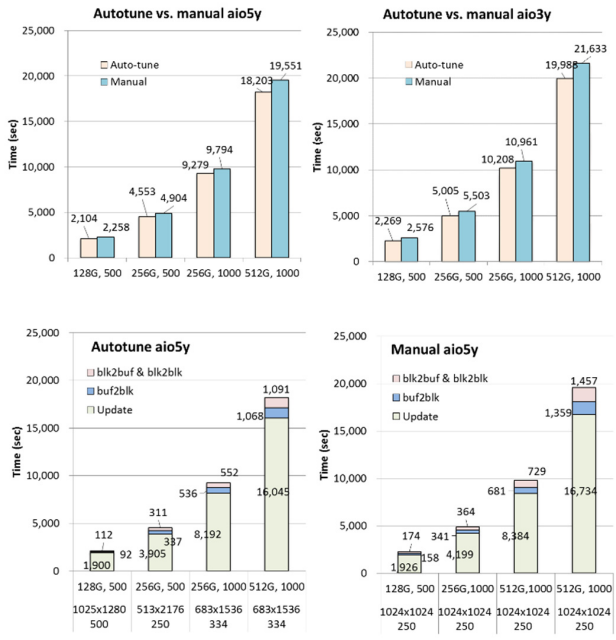

Figure 22. *Manual selection* vs. *Blk-Tune* for various-size problems on crest4 system.

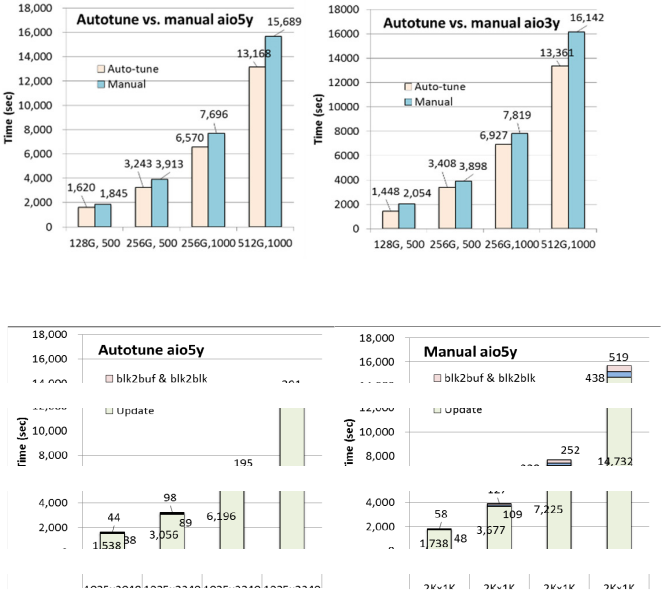

Figure 23. *Manual selection* vs. *Blk-Tune* for various-size problems on crest6 system.

D. *Performance of Blk-Tune vs. manual selection*

According to platform information and problem settings, *Blk-Tune* automatically selects optimal blocking sizes, *Block array* (*bx, by, bz, bt*) and *iBlock* (i*x, iy, iz)*. Table 2 shows examples of selected parameters for 128GiB and 256GiB problems in each platform, creat0, crest4, and creat6.

The upper graphs in Fig. 22 show the execution times obtained by using *Blk-Tune* and *manual selection* for aio3y and aio5y for problems of various sizes (128–512 GiB) and iterations (500–1000) using crest4 server (16 cores, 64GiB DRAM). The problem sizes corresponds to 2-time, 4-time, and 8-time larger than the DRAM capacity on crest4. The charts in the lower part of Fig. 22 show the time components obtained using *Blk-Tune (auto-tune)* and *manual selection* for aio5y. *Blk-Tune* selects optimal parameters that decrease the IO times to 58%-78% of the times in manual selections.

The upper charts in Fig. 23 show the execution times and time components for the same problems using a different server, i.e., crest6 (20 cores, 128 GiB DRAM). On this platform, *Blk-Tune* also achieves a performance improvement. The each time component, flash IO and update, in each problem in Fig. 23 is smaller than the counterpart in Fig. 22, because crest6 has faster CPUs and a flash device.

Selected *bz and iz are* multiples of the number of cores, *Nc*, in each platform to achieve efficient multi-thread work-share and parallel processing in the *z*-dimension. In many cases, *bz* is selected to be a multiple of *iz*. By considering the effect of division schemes and boundary conditions, an evaluation of the tradeoff between spatial and temporal blocking sizes under fixed-size of DRAM can be conducted more precisely.

As shown in Table II, *by* which is smaller than *bz* is usually selected, because of the internal algorithm procedure in aio3y

and aio5y. Dividing a *Buffer array* in the y-dimension, i.e., ny/by, results in a smaller amount of flash IO compared to when dividing it in the z-dimension, i.e., *nz/bz*. In the case where the array is divided in the y-dimension, memory copy, instead of flash IO, is used for boundary data forwarding from a current updated block to a neighboring block in the y-dimension to be updated next. Thus, *Blk-Tune* prefers a smaller value of *by* compared to *bz*, which reduces the amount of flash IO.

TABLE II. PARAMETER EXAMPLES SELECTED BY BLK-TUNE

Problem size	nx	ny	nz	nt		
128GiB-problem	2048	2048	2048	500		
Server (DRAM, cores)	bx, ix	by	bz	bt	iy	iz
crest6 (128GiB, 20)	2048	1025	2048	500	1	160
crest4 (64GIB, 16)	2048	1025	1280	500	1	128
crest0 (32GiB, 8)	2048	513	1152	250	1	64

Problem size	nx	ny	nz	nt		
256GiB-problem	2048	2048	4096	1000		
Server (DRAM, cores)	bx, ix	by	bz	bt	iy	iz
crest6 (128GiB, 20)	2048	1025	2240	500	1	160
crest4 (64GIB, 16)	2048	683	1536	334	1	128
crest0 (32GiB, 8)	2048	513	832	250	1	64

E. Auto-tuning time in Blk-Tune

The current parameter search implementation is primitive and naïve and leads to an increase in auto-tuning time in proportion to the problem size as shown in Fig. 24. There are many ways to speed up the auto-tuning times in the *Blk-Tune* system. However, even using the current version, the auto-tuning time is negligible as compared to stencil computation times, e.g., 0.1% in 128–512 GiB problems.

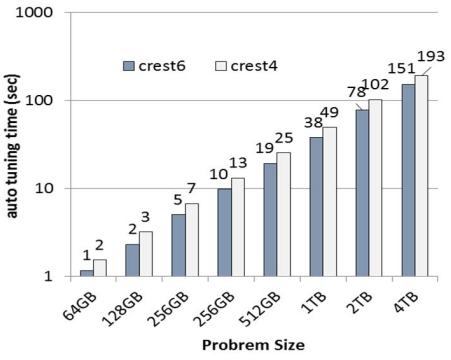

Figure 24. Tuning time in *Blk-Tune* for various-size problems in crest6 and crest4.

VII. CONCLUSION

This paper proposes *Blk-Tune,* a runtime automatic blocking-parameter tuning system designed for flash-based out-of-core stencil computations. It is based on a global optimum search using a cost function reflecting the amount of IO traffic to flash devices. It minimizes the amount of data transferred between a flash device and DRAM, which is the most dominant factor affecting the performance of out-of-core algorithms using flash. The use of explicit highly parallel asynchronous IO to a flash device together with auto-tuning realizes the achievement of maximum performance of large-scale stencil computations in different platform and application settings.

We are currently extending this configuration to accommodate the use of various stencil algorithms and for flash-equipped cluster systems. Future work also includes the development of more general APIs for various kinds of stencil computations.

ACKNOWLEDGMENT

The author would like to thank Prof. Toshio Endo, Dr. Yukinori Sato, and Dr. Shimpei Sato for their discussions and advice on temporal blocking stencil algorithms, and Mr. Hideyuki Tan for his contributions in terms of technical support, especially in building experimental environments.

REFERENCES

[1] H. Midorikawa, H. Tan, and T. Endo, “An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations,” Proc. 2014 Int. Conf. on High Performance Computing and Simulation IEEE-HPCS2014, Jul. 2014, pp. 268-277, doi: http://dx.doi.org/10.1109/HPCSim.2014.6903695.

[2] H. Midorikawa, “Using Flash SSDs as Main Memory Extension with a Locality-aware Algorithm,” Non-Volatile Memories Workshop, Mar. 2015.

[3] H. Midorikawa and H. Tan, “Locality-Aware Stencil Computations using Flash SSDs as Main Memory Extension,” Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing CCGrid2015, May 2015, pp. 1163-1168, doi: http://dx.doi.org/10.1109/CCGrid.2015.126

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

[5] Portable Hardware Locality (hwloc). https://www.open-mpi.org/projects/hwloc/

[6] A. Badam, "How Persistent Memory Will Change Software Systems," IEEE Computer, vol. 46, pp. 45-51, Aug. 2013.

[7] K. Sudan, A. Badam, and D. Nellans, "NAND-Flash: Fast Storage or Slow Memory?," Non-Volatile Memories Workshop, Mar. 2012.

[8] SanDisk ioMemory, http://www.sandisk.com/enterprise/pcie_flash/fusion-iomemory

[9] Samsung http://www.samsung.com/semiconductor/products/flash-storage/

[10] Y. Zhang and F.Mueller, ”Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters”, IEEE Trans. Parallel and Distributed Syst., vol. 24, pp. 417-427, May 2012.

[11] Y. Luo, G. Tan, Z. Mo, and N. Sun, “FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model,” ICS '15 Proc. 29th ACM Int. Conf. on Supercomputing, 2015, pp. 187-196, doi: http://dx.doi.org/10.1145/2751205.2751214.

[12] J. D. Garvey and T. S. Abdelrahman, “Automatic Performance Tuning of Stencil Computations on GPUs,” Parallel Processing (ICPP), 44th Int. Conf., 2015, pp. 300 – 309, doi: http://doi.ieeecomputersociety.org/10.1109/ICPP.2015.39.

[13] P. Basu, M. Hall, and S. Williams, “Compiler-Directed Transformation for Higher-Order Stencils,” IEEE Int. Parallel and Distributed Process. Symp. (IPDPS), 2015, pp. 313-323,

[14] H. Jordan, et al., “A multi-objective auto-tuning framework for parallel codes,” Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC ’12), Nov. 2012, pp. 1 - 12.