Optimization of Memory Allocation in VSIPL

Jinwoo Suh, Janice O. McMahon, Stephen P. Crago, and Dong-In Kang
University of Southern California – Information Sciences Institute
3811 N. Fairfax Drive, Suite 200, Arlington, VA 22203
{jsuh, jmcmahon, crago, dkang}@isi.edu

Abstract

In this paper, we propose an efficient memory allocation algorithm for Vector, Signal, and Image Processing Library (VSIPL) standard. This algorithm improves the efficiency of key VSIPL functions by exploiting fundamental properties of VSIPL objects during the allocation of memory. We implemented and tested our algorithm using the VSIPL reference implementation and measured results on representative VSIPL functions.

Introduction

Vector, Signal, and Image Processing Library (VSIPL) standard has been proposed to support portable, high performance application programs [1]. A first draft version 1.2 for C language was proposed in 2006. The library provides many operations such as vector addition, matrix multiplication, and 2D-FFT.

Input and output data for VSIPL functions are stored in "blocks" which represent areas in memory. Abstract data types such as "block" and "view" objects are provided in VSIPL for storing and accessing data, respectively [1]. In this paper, we optimized the memory allocation for these objects and measured the resulting performance improvement on VSIPL functions. In this abstract, results for matrix addition are presented.

Platform

In our experiments, we installed a reference implementation called the TASP_VSIPL_Core_Plus library [2] implemented by Randoll Judd, and used it as a baseline for our work. This implementation of VSIPL was developed for to provide a reference for the official specification and focused on functionality rather than performance. The hardware used in our experiments is an Intel Core 2 E6400 dual-core machine running Redhat linux version 2.6.9-5. The machine runs at 2.1 GHz with 2 GB main memory.

Memory allocations in VSIPL

In a reference implementation of VSIPL, a malloc() is used to allocate memory for data objects. Whenever these objects are not needed, free() is called. However, these calls are expensive in terms of cycles since they are handled by the operating system (OS) and include overhead for management, bookkeeping, etc. As more and more calls are used, the heap space can be easily fragmented, making it harder to find space for malloc().

Therefore, it would be beneficial to have a memory allocation scheme tailored to VSIPL that avoids the overhead of malloc(). In this implementation, we put higher priority on overall function performance than the efficient use of memory, and therefore we chose a first-fit-based algorithm.

Our memory allocation scheme exploits a few key properties of VSIPL data blocks, block objects, and view objects. Firstly, the block object and view object sizes are the same. Secondly, the data block, block object, and view object are created at the beginning of executions usually as a group. Finally, the objects live until the end of a program. In other words, data block, block object, and view object are not created and destroyed dynamically, but instead exhibit quasi-static behavior that can be exploited..

Based on these observations, we devised the following algorithm for view object and block object:

1. Allocate memory space for view objects and block objects separately.
2. Allocate memory space for indexes for view and block objects. Each word or bit in the index corresponds to a view or a block in the view and block memory space. If an index is 1, it means the corresponding block or view is being used. Otherwise, the block and view space is not being used.
3. The create operation for blocks or views, scans indexes from beginning to find an index that has zero value. Then, the start address of the corresponding block or view is returned. If a single bit is used for the index, 32 indexes can be checked at the same time by checking a single 32-bit word. After this, there are extra steps to pinpoint the exact location of the empty space.
4. The destroy operation for blocks or views calculates the corresponding index from the block address and the starting address of the reserved space for view and block objects. Then, the corresponding index is reset to zero. This freed space and corresponding index can be reused by the next create operation.

By using this algorithm, the overall create and destroy cost is significantly reduced by avoiding general purpose and time-consuming memory allocation operations and instead using allocation functions that exploit the specific properties of blocks and views such as fixed sizes.

We also implemented a second algorithm for variable sized data blocks. Since the sizes of input and output data for VSIPL operations are not fixed, a fast memory allocation scheme needs to be provided for variable data allocation. The following algorithm is used for variable data allocation for VSIPL.

1. Allocate memory space for data blocks.
2. Allocate memory space for indexes for data block. Each index is a structure containing three fields: a) a valid bit. If the valid bit is 1, it means the index is being used. Otherwise, the index is not being used. b) start field that contains start address of the corresponding data block. c) an end field that contains last address of the corresponding data block.
3. When a new data block needs to be allocated, the indexes are scanned one by one from the beginning. When an index whose valid field is zero is found, it checks the available space by calculating the difference between the start filed of the next index and the end field of the previous index. If the size is larger than the requested size, then, the index is updated such that the valid field is set to 1 and start and end fields are updated. Then, the start address is returned.
4. If a data block is freed, the index is scanned to find an index that has start address matching the start address of the space to be freed. If found, the valid field of the index is reset to zero.

Implementation Results

Fig. 1 shows the number of cycles for each operation for 8 by 8 matrix addition. The figure shows that the numbers of cycles are reduced for most of the operations. The most reduction is in blockbind operation where 5.6 times of speedup was achieved.

The big saving in the blockbind is somewhat compensated for by increased number of cycles at mbind stage where the number of cycles is actually increased by 17.6K cycles. The increased number of cycles is due to the fact that the data had to be used in the mbind stage. To use the data in mbind stage, the data are moved from memory to processor. In the straightforward algorithm, the data is moved to cache in the blockbind stage. However, the increased number of cycles in mbind stage is much less than the reduced number of cycles in blockbind stage, and as a total, the number of cycles is reduced significantly.

Fig. 2 and Fig. 2 show the numbers of cycles for total and VSIPL object related operations for matrix addition and matrix maximum, respectively using straightforward algorithm and new algorithm. The figures show that the numbers of cycles for VSIPL related operations are reduced significantly and remain constant for any matrix sizes. As the matrix sizes increases, the number of cycles for matrix addition and maximum are increased linearly and the relative difference becomes smaller.

Conclusion

In this paper, we developed an efficient algorithm for memory allocations in VSIPL. By exploiting the properties of VSIPL data objects such as blocks and views, the new algorithm achieved up to a factor of 3.5 speedup over straightforward algorithm. Since block and view objects are fundamental to VSIPL, the new algorithm can be used for any operation in VSIPL, and we expect to see similar performance improvements in other functions.

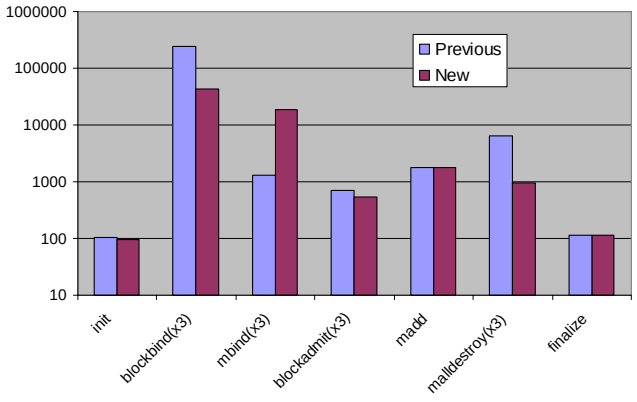

Figure 1. Number of cycles for each operation for stack data allocation

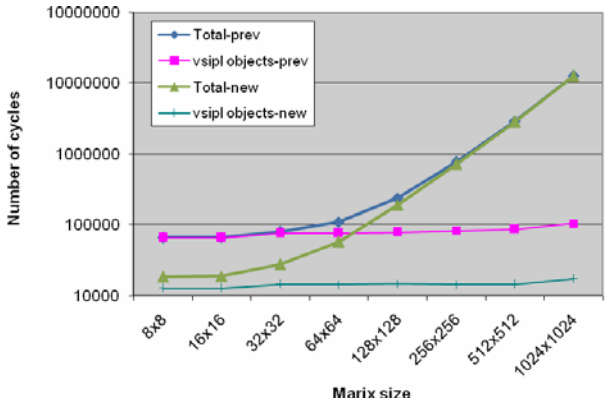

Figure 2. Number of cycles for each operation for stack data allocation

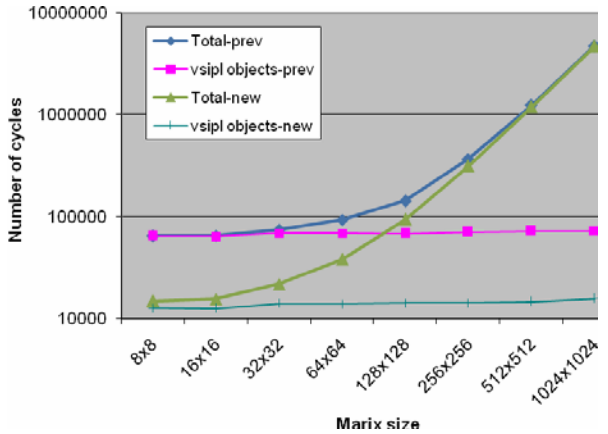

Figure 3. Number of cycles for each operation for heap data allocation

References

[1] D. Schwartz, et. al, "VSIPL 1.2API," http://www.vsipl.org, April 2006.

[2] R. Judd, "TASP_VSIPL_Core_Plus library," http://www.vsipl.org/software, 2007.