THE EFFECT OF REORDERING MULTI-DIMENSIONAL ARRAY DATA ON CPU CACHE UTILIZATION

by

Alireza Ghane
B.Sc., Sharif University of Technology, 2005
M.Sc., Sharif University of Technology, 2007

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Alireza Ghane 2013
SIMON FRASER UNIVERSITY
Spring 2013

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be
reproduced without authorization under the conditions for "Fair Dealing."
Therefore, limited reproduction of this work for the purposes of private study,
research, criticism, review and news reporting is likely to be in accordance
with the law, particularly if cited appropriately.

APPROVAL

Name: Alireza Ghane

Degree: Master of Science

Title of Project: The Effect of Reordering Multi-Dimensional Array Data on CPU Cache Utilization

Examining Committee: Dr. Hao (Richard) Zhang
Chair

Dr. Torsten Möller,
Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Arrvindh Shriraman,
Assistant Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Alexandra (Sasha) Fedorova,
Associate Professor, Computing Science
Simon Fraser University
SFU Examiner

Date Approved: January 4, 2013

Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011

Abstract

Memory, as a shared resource, has always been a high latency and bandwidth limited bottleneck of the execution pipeline in multi-core systems. This project analyzes data reordering in multi-dimensional arrays for a more efficient memory allocation method to improve cache utilization and reduce memory access bandwidth. While single-threaded run-time improvements are limited, we demonstrate up to 30% improved run-time and energy consumption in multi-threaded applications when the processing cores are competing for cache space and memory bandwidth.

Contents

Approval ii

Partial Copyright License iii

Abstract iv

Contents v

List of Tables vii

List of Figures viii

Preface x

1 Memory Architecture and Performance Analysis **1**

1.1 Virtual Memory and TLB 1

1.2 Cache Memory and Architectures 2

1.3 Memory-Cache Mapping 3

1.4 Virtual vs. Physical Indexing Cache 4

1.5 Hardware Performance Monitoring 5

1.6 Chapter Summary 6

2 Software Solutions for Cache Optimization **9**

2.1 Approaches to Cache Optimization 9

2.2 Data Reordering 10

2.3 Space-Filling Curves and Morton Order 12

2.4 Index Calculation 15

2.4.1 Morton Index with Dilated Integers . . . 16
2.4.2 Morton Index with Look-up Tables . . . 21
2.5 Morton Address Update . . . 21
2.6 Hybrid Solution . . . 23
2.6.1 Hybrid Morton . . . 23
2.6.2 Dimension Shuffle Multi Block . . . 24
2.7 Chapter Summary . . . 26

3 Experiments **27**
3.1 Experiment Setting . . . 28
3.2 Results . . . 29
3.2.1 Data Load Requests . . . 29
3.2.2 On-Core Cache Utilization/Bandwidth . . . 30
3.2.3 Off-Core Data Access and Main Memory . . . 31
3.2.4 TLB and Paging Effects . . . 35
3.2.5 Overheads and Run-time . . . 36
3.2.6 Energy Consumption . . . 37
3.3 Chapter Summary . . . 39

4 Conclusion and Future Work **41**

Appendix A Hardware Performance Counters **43**
A.1 Performance Events . . . 43
A.2 Performance Metrics . . . 45

Bibliography **46**

List of Tables

3.1 The cache and memory properties for Intel SandyBridge i5-2300. [4, 3, 1, 2] . 28

A.2 The list of performance metrics we used, with their calculation formula. Metrics in the same block are measured in one run. 45

List of Figures

1.1 Memory Paging for a typical machine with a 32-bit address bus, and 4KB page size. The virtual page address is translated to its physical address using lookup tables. If the table entry does not exists in the fast TLB cache, the page table entry in main memory should be accessed. 2

1.2 4-way set-associative cache with 256 sets, and a 4-word block size. The first 2 bits of the address separate the words in each block. The next 8-bit segment is used to specify the candidate set in the cache, highlighted in blue. Of the 4 possible candidate lines, the one whose tag agrees with the remaining 22 bits of the address is selected. If no cache line in the set satisfies this condition, a cache-miss occurred and the data should be requested from higher level memory. [18] 5

2.1 Data ordering in 2D for 8×8 data: a) Scanline order; b) Block order with a block size equal to 16 data points. 12

2.2 Three iterations of a Peano curve construction. [5] 13

2.3 Z-order and U-order; different generations of Morton order. 13

2.4 Double-Gray Morton: Optimal locality distortion ordering based on Morton order. The ordering in each 2×2 cell is reversed to reduce the average distance. The same pattern repeats in higher levels. 15

2.5 SpecialAdd function is implemented by modifying a serial adder to have n shifts for each carry bit instead of one shift. 22

2.6 3-Level block indexing for point (X,Y) in a 2D array of size 256×256: Top: Regular Scanline; Bottom: Dimension Shuffle. 24

2.7 3-Level block indexing for point (X,Y,Z) in a 3D array of size $256 \times 256 \times 256$: Top: Regular Scanline; Bottom: Dimension Shuffle. 24

2.8 3-Level block indexing for point (X,Y,Z,W) in a 4D array of size $64 \times 64 \times 64 \times 64$: Top: Regular Scanline; Bottom: Dimension Shuffle. 24
2.9 Index increment on the first dimension for 2D arrays. S is the initial dimension-shuffle index for the point (S_X, S_Y) and S' is the updated index for the point $(S_X + 1, S_Y)$. 25

3.1 Average size of data (in GB) loaded on each CPU core; or equivalently, the data requested from the L1 cache of each core. While the graph represents measurements for four threads, there is no difference for 1, 2, 3, or 4 threads. 29
3.2 Average data transfers from L2 to L1 cache at each core; or equivalently, the data requested from L2 cache of each core, for 1/2/3/4-threaded line integral algorithm; First row: data size; Second row: average bandwidth. 30
3.3 Average size of data transferred from L3 to L2 on each core; or equivalently, the data requested from L3 cache from each core. Top row: Single-threaded; Bottom row: quad-threaded. 32
3.4 Average bandwidth used to transfer data from L3 to L2 for each core; or equivalently, the data bandwidth requested from L3 cache by each core. Top row: Single-threaded; Bottom row: quad-threaded. 33
3.5 L2 cache requests: L1/LFB cache misses, which needs access to L2 and higher. 34
3.6 The performance of L2 and L3 caches: First row: Data loaded from L2 as source; Second Row: Data loaded from L3 as source. 35
3.7 L1 DTLB miss rate (number of misses per instruction), for the single-threaded line integral algorithm. 36
3.8 Total number of retired instructions. Single-threaded line integral. 37
3.9 Total run-time in seconds for line integral computation. Top row: single-threaded; Bottom row: quad-threaded. 38
3.10 Total Power (W) used by the processor package for line integral computation. Top row: single-threaded; Bottom row: quad-threaded. 39
3.11 Total energy (J) used by the processor package for line integral computation. Top row: single-threaded; Bottom row: quad-threaded. 40

Preface

Cache is the solid popular solution to hide main memory latency from the processor. With more and more compact chip technologies, processor speed reached more than 100 times that of memory. Cache is a small and fast memory, with speed closer to the processor, which reduces the average waiting time for memory accesses. Data required from the main memory is either prefetched to the cache in advance, or temporarily stored in cache instead to prevent the slow memory write and read process.

Although cache memory is introduced to improve code execution run-time, reducing off-chip requests has other benefits as well. Keeping data transmissions on the chip reduces energy consumption as well as data traffic on the system bus. Although these achievements had not been real concerns for early computers, they become more and more important every day with the rise of multi-processor and multi-core systems, especially on battery-powered devices.

Cache memory is usually defined in several levels. The first level, called L1, is the closest to the processor core, with the speed close to core registers. Higher level caches are bigger with higher access latency due to their larger distance to the core. In a multi-core system, there is usually another level of cache shared among all cores on the processor package to prevent off-package operations. Cache memory and other on-chip memory circuits occupy more than 50% of the chip [28], and spend 40% and sometimes up to 70% of the CPU power [12].

Since cache is a very limited resource compared to the main memory, efficient use of it has always been a challenge. There are a variety of approaches to get the best use of this fast memory. Prediction and prefetching is a well established technique to load data in the cache before it is requested by the execution pipeline. To achieve best performance, the prefetcher should be able to predict the next required memory location, which is a challenge

for general purpose processors.

Multi-core processors is the current technical breakthrough in computer systems design in recent years. While the processor clock speed is at its technological limit, modern processors take advantage of on-chip parallelization for another level of outstanding speedup. The idea of having several simple cores in a package, first started in GPUs for image rendering, is now extended to multi-core general purpose processors that can be found in any device. This technology is not only limited to high performance systems, but they are used in desktops, laptops, tablet computers, and even smart-phones and embedded systems. From mid-range to high-end Intel Core©, to Intel Atom© and ARM Cortex-A© designed for light low-power tablets and smart-phones, a multi-core architecture is a solution to achieve the desired performance.

Despite increased complexity in multi-core architectures there are benefits. Due to parallel code execution, the efficient use of shared resources become an increasing concern. Memory as a shared resource with high latency and limited bandwidth is not an exception. Efficient use of on-core cache can reduce requests from the shared memory and prevent memory bandwidth congestion. As Liu et. al. [26] reported, prefetching might not be an efficient way in such systems. They claim that prefetching efficiency diminishes rapidly when multiple cores access the memory simultaneously, straining the shared bandwidth. Preventing the simultaneous access is an NP-hard scheduling problem [11] and can be just partially optimized in real-time.

Code and data locality is the idea behind cache usability. When a location of the memory is accessed as code or data, it is likely that it will be accessed again due to loops in the program, functions recalls, and reuse of variables. This property is called *temporal locality*. Besides the reuse of the same memory location in time, there is also *spatial locality* due to sequential code flow and continuous data blocks. Spatial locality refers to the fact that neighboring memory locations of the accessed address in memory have a high chance of getting accessed in the near future.

Spatial and temporal locality are the basic assumptions for cache hardware design and prefetcher optimization. There are software solutions to improve locality in the code and data, which helps the prefetcher and the basic cache architecture to work efficiently. Temporal locality is mainly based on the flow of the code and the way the variables and data structures are used. Spatial locality is determined by the mapping of data structures to main memory. To get the best use of cache, data fields that are likely to be used at the

same time period should be mapped close to each other in memory.

The spatial locality property cannot be completely satisfied for high-dimensional data structures. Given a data item in an n-dimensional array, it has $2n$ direct neighbors but there are only 2 neighboring locations in 1D memory mapping. The same argument is true for larger neighborhood blocks. This work studies efficient mappings of n-dimensional arrays to linear memory such that spatial locality is preserved as much as possible with minimum overhead. We study Morton ordering, which is used for matrix multiplication [40]. In addition, we introduce two hierarchical orderings based on array block ordering. We show that the full spatial locality optimization is not required for optimal cache utilization. Our 3-level block ordering called *Dimension Shuffle Block Ordering* has a similar performance to Morton ordering with fewer index calculations. We also study other options for Morton index calculation, especially for higher dimensions.

Beside cache utilization, we measure run-time and energy improvements on multi-core systems by data reordering. Our sample application is line integral calculation in a 2 to 4-dimensional hyper-cube with different data sizes and different ordering methods. We tested two implementation of Morton ordering, hybrid Morton-scanline, and a hierarchical block ordering with dimension shuffling. While the most efficient Morton ordering achieves a speedup up to 39% in 2D, 3% in 3D, and 6% in 4D over scanline, our dimension shuffle solution outperforms scanline with 44%, 19% and 12% respectively. Our energy consumption also reduced by up to 44% for 2D, and 14% for 3D and 4D. We also have more than an order of magnitude improvement on TLB [1] misses, which shows a high potential for a reduction of page faults. As a result, we expect more dramatic improvements on systems with smaller main memory or applications with larger data sizes, where the data need to be swapped from RAM to hard disk.

[1] Translation Lookaside Buffer: cache memory dedicated to the memory address translation table.

Chapter 1

Memory Architecture and Performance Analysis

This chapter provides the basic concepts of memory architecture in modern processors. The concepts defined here are fairly general for most CPU-based systems including servers, desktop computers, laptops and mobile smartphones.

1.1 Virtual Memory and TLB

Since the mapping between hard disk and main memory is not one-to-one, a method is required to define the actual physical address for a program. The goal of virtual memory space is to make this mapping transparent to the programmer, and to provide a unified addressing space to an executable. Every time a file is loaded or a memory segment is initiated, an lookup table entry is added, enabling virtual to physical address translation. The address translation is done automatically afterwards during the execution of the program.

To make this translation more efficient, the addressing space is divided into blocks of fixed size called *pages*. Typically, each page has a fixed size of 4/8/16KB. The memory address is broken into *page address* and *page offset*. In virtual to physical memory mapping, the page offset remains unchanged, while the virtual page address is translated to the physical page address. The two parts together make up the full physical address. This whole process is summarized in Figure 1.1

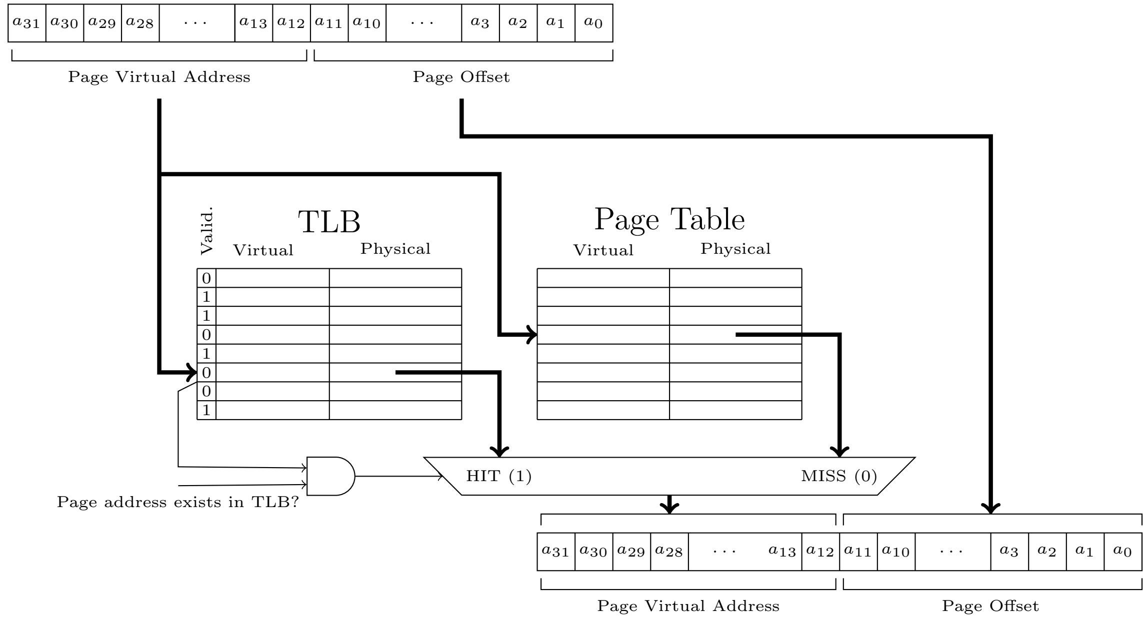


Figure 1.1: Memory Paging for a typical machine with a 32-bit address bus, and 4KB page size. The virtual page address is translated to its physical address using lookup tables. If the table entry does not exists in the fast TLB cache, the page table entry in main memory should be accessed.

The address translation is done using a look-up table called *page table*. Since this page table is used in each memory access, efficient and fast implementation of it is one of the key factors in code execution performance. The CPU's memory management unit stores the recently used page table entries in a specialized fully-associative cache, called *Translation Lookaside Buffer* (**TLB**).

1.2 Cache Memory and Architectures

Since code execution on the processor is much faster than the data transfer speed from the main memory, execution speed is limited by memory latency. To reduce the effect of memory latency, faster levels of intermediate memory, called *cache*, are introduced. Cache is usually inside the CPU package to reduce slow off-chip communications. Due to limited area on the processor chip, cache memory is much smaller than main memory. There can be different (hierarchical) cache levels, also referred to as L1, L2, etc. Typically, low levels

of cache are smaller with higher speed. The lowest level can be close to CPU registers. The higher levels of cache are larger and cheaper in comparison.

Every time the processor wants to access data in the memory, the request is sent to L1 cache. If the data exist in the cache (*cache hit*), it is loaded to the processor with minimal latency. If the data is not in L1 cache (*cache miss*), it is requested from the next level cache (L2), which is larger and slower. The same scenario repeats between L2 and L3 cache (if it exists), the last cache level and main memory, and between main memory and the disk.

1.3 Memory-Cache Mapping

The cache memory is divided into equal size blocks called **cache line**. The typical size of the cache line is 32 to 256 Bytes. A cache line is the smallest block of data that is transferred from and to memory or next level cache at each data transfer. Since cache is smaller than the main memory, a mechanism is required to help the processor finds the requested data from the cache if it exists. This mechanism should also include verification for determining the exact memory location stored in the cache as well as data validity.

The simplest memory-cache mapping is called **direct mapping**. In direct mapping, each memory block has only one candidate line in the cache. If the processor needs to address this memory location, only one cache line should be checked. Since the cache memory is smaller, obviously this mapping cannot be one-to-one. To find out which memory locations are stored in a cache line, among all the locations mapped to the same cache line, a tag is attached to the data stored in each line. The location of the cache line, together with the tag determines a unique location in memory. The typical hardware implementation of direct mapping is to use the bit segments of the main memory address as the cache line address and the tag. For example, suppose the word size is 1 byte and the cache line is 16 bytes, for a 1KB cache with direct mapping. Starting to count from the least-significant bit, the first 4 bits of the address is the offset inside the cache line, showing which word among the 16 words in the cache line is the target. Since the cache size is 1KB, it contains 64 cache lines. So, the next 8 bits are used as the cache line address. The remaining bits should be stored as a tag for each cache line. To maximize the distance between the addresses mapped to the same cache line, least significant bits are used as the cache address while the remaining ones define the tag.

Another extreme approach is to have the possibility to store each memory location

anywhere in the cache. This is called **fully associative cache**. In such a scenario, all physical address bits should be used as the tag and the full cache should be checked for the requested data. On the other hand, the cache utilization would increase and the likelihood of cache misses would decrease.

The engineering trade-off between these two extreme cases is called **set-associative cache**. In set associative cache, each memory location has a small set of possible candidates in the cache. For each memory request, only a subset of cache lines should be searched. The set is determined with the bit segments of the address, similar to direct mapping. The remaining bits should be attached to the cache block as a tag. Clearly, there are smaller number of sets in set-associative cache compared to the blocks in direct mapping cache with the same size. The memory address of the data in the candidate cache line can again be verified with its tag, which is larger than the direct mapping due to the shorter bit segment that determines the cache location.

Figure 1.2 shows an example hardware implementation of a 4-way set associative cache. When the set size increases, a larger possible set of cache locations should be checked for the requested data. It increases access latency and the size of the cache control circuits, as well as power consumption. Zhang et. al. [44, 43] claim that a direct mapping cache uses 30% less power than a 4-way set-associative cache with the same size. The typical implemented size of the set in modern processors is between 1 and 16, depending on technology, cache level, and power constraints.

1.4 Virtual vs. Physical Indexing Cache

As explained in Section 1.1 the memory address that the executable requests is different from the physical memory address accessed in main memory. This translation can be done at any cache level depending on the CPU architecture. For example, if L1 cache has virtual and L2 cache has physical addressing, any miss for the L1 cache leads to a TLB access for the address translation, along with an L2 request for accessing data. On the other hand, if the data exists in L1, no virtual to physical translation is needed and TLB access is not required.

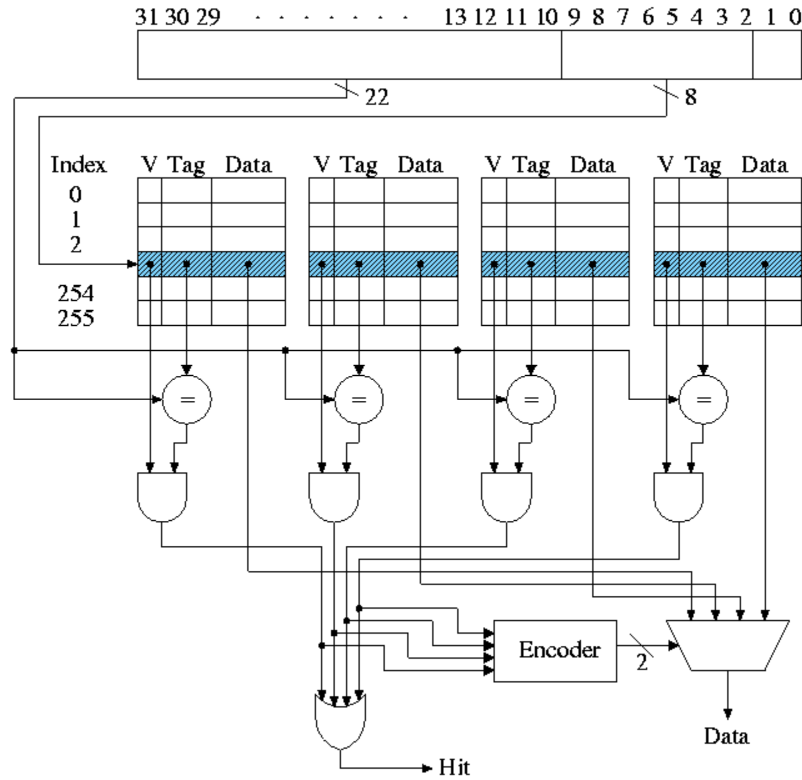


Figure 1.2: 4-way set-associative cache with 256 sets, and a 4-word block size. The first 2 bits of the address separate the words in each block. The next 8-bit segment is used to specify the candidate set in the cache, highlighted in blue. Of the 4 possible candidate lines, the one whose tag agrees with the remaining 22 bits of the address is selected. If no cache line in the set satisfies this condition, a cache-miss occurred and the data should be requested from higher level memory. [18]

1.5 Hardware Performance Monitoring

There are generally two ways of performance measurements on a processor: software profiling and hardware monitoring. In software profiling, the profiler simulates the system running the program and estimates the requested metrics. In hardware profiling, performance metrics are measured on the real system running the code. Typical modern processors provide mechanisms for hardware monitoring with minimal overhead on the running code.

Hardware monitoring on a modern processor is done by special registers known as **Hardware Performance Counters**. These registers can be configured to count special events such as retired instructions, number of cache misses and so on. Every time the event happens, the counter increases without any effect on the code execution. When the counter reaches a threshold, an interrupt routine is initiated to accumulate the counter value with the previous data and reset the counter for the next round.

On the other hand, it is often very hard, if not impossible, to get local information form these counters, such as profiling a small part of the code inside a loop or separating the exact effect of one thread among hundreds of threads running on the system.

The number of special registers used as hardware performance counter depends on the processor type. The types of events they can measure is also different from processor to processor. The short list of hardware performance counters we used in our simulations for the Intel i5-2300 (SandyBridge© Cores) processor can be found in Appendix A.

1.6 Chapter Summary

Cache memory is used in most of the modern processors to reduce the average latency of data access in the system. Lower level cache is faster and smaller while cache levels closer to the main memory are slower and larger in size. The mapping between cache and main memory locations can be a direct mapping or set-associative, depending on the number of possible cache lines the data can be stored in for each location of main memory.

To give the system the flexibility to load code and data at any location of the memory for a generic program, the concept of virtual memory is defined. Virtual memory is mapped to the physical memory using a page table. The fixed part of the cache reserved for caching page table items is called Translation Lookaside Buffer, or TLB. The page table or TLB should be accessed at the first cache level that uses physical memory addressing.

There are three types of memory access from the processor: instruction access, data access, and TLB access. At all three levels cache memory can be used to reduce latency. TLB cache is usually a fixed separate cache while the instruction and data cache can be shared. Usually the first level instruction and data cache are separate while the rest are shared.

In a multi-core system, the cache can also be specific to a core, or shared among all cores. When one core requests data that is available in another core's cache, the transmission

is usually done without going to the next level cache. This scenario usually happens in **exclusive** cache structure, where the low level cache may include data that is not in the higher level cache. The other scenario is **inclusive** cache. In inclusive caches, the higher level cache has a full copy in the lower cache. For example, inclusive L2 cache means that every data available in L1 has a copy in L2 as well. Handling data in a multi-core system is easier with inclusive cache but it wastes a portion of cache due to data replication.

The following list shows all the technical terms defined in this section and used later in the thesis:

- **Page**: A page is the unit size of data (usually of size 4, 8, or 16 KB) that transfers from disk to the main memory. While the address offset inside the page remains the same, the location of the block requires an address translation from its assigned virtual (in the program environment) to physical address (in memory).

- **Page Fault**: The event of missing a requested page in the memory. When a page fault happens, the requested data is missing from memory and should be loaded from the disk.

- **Page Table**: The look-up table used for mapping pages into main memory. In other words, it is the table used for mapping a virtual address to a physical address, which is decided by the operating system.

- **TLB**: A Translation Lookaside Buffer is the specialized portion of cache for caching page table items.

- **Cache Line**: The unit block in cache that is accessed, filled, and replaced.

- **Cache Hit/Miss**: When the requested data is available in the cache we say a cache hit happened. Cache miss is the event of missing the requested data in the cache, which requires access to the next level memory.

- **Set-Associativity**: A mapping policy between addressable space and the cache. In an n-way set-associative cache, there are n candidate cache lines for a line in the memory.

- **Cache Indexing**: The addressable space the cache deals with. In virtually indexed cache, the virtual address is used as the address. For physically indexed cache, the

virtual address should be translated to the physical address before locating the data in the cache.

- **Inclusive/Exclusive Cache**: An inclusive cache retains a copy of the lower level cache. In other words, if the data is thrashed in the cache, it is also thrashed from the lower level cache. An exclusive cache might thrash a cache line without removing it from other caches.

Chapter 2

Software Solutions for Cache Optimization

2.1 Approaches to Cache Optimization

Cache memory is an expensive part of the modern processors. A large cache adds to the complexity and price of the processor and efficient use of it is the goal of many researchers and engineers. The key idea for cache optimization is taking advantage of data locality: **Spatial locality** is a property for which data or code segments that are close to each other in memory are more probable to be used together within a short period of time. Data or code segments have **temporal locality** when their currently accessed memory location is more probable to be used again in the near future compared to any other location. Spatial locality usually applies to sequential parts of the code and small arrays while temporal locality applies to small loops and temporary variables in the code. The basic use of temporal locality is the LRU (Least Recently Used) cache replacement policy for cache lines in each set, while spatial locality is used by prefetching policies.

There are many different approaches to the cache optimization problem. Some are embedded inside the processor production with little user control. The most popular class of this type, besides the replacement policies, is the prefetching techniques that try to predict the data that will be requested in the future and prefetch it beforehand [42, 31]. Although prefetching methods have, in general, been successful in reducing the latency of memory accesses, Srinath et. al. [36] argue that the prefetcher can reduce performance if not

used properly. Prefetching data has the potential to cause memory bandwidth waste and unnecessary cache conflicts, especially when the cache is shared between several independent threads or an accurate prediction is not possible. It is also not always a wise choice when the energy consumption is a concern.

There are also a variety of pure software approaches to this optimization problem. Kessler [22] claims that an intelligent page mapping by the operating system can prevent up to 30% of cache conflicts in a physically-indexed cache. Perarnau et. al. [33] propose a cache partitioning idea for HPC (High Performance Computing) applications. They believe that isolating each critical data structure into a separate cache partition opens opportunities for significant optimization strategies.

Beside research publications, there are a vast number of patents on this issue. Archambault et. al. of IBM [7] introduce a compiler with cache utilization optimization by processing the call graph of the application and life cycle of the data structures. Larus et. al. of Microsoft [24] approach the problem by reordering the fields inside data structures and find the order with the best cache utilization. Chilimbi et. al. [13] improve cache utilization with data structure partitioning into hot and cold fields, and optimize the mapping for the hot fields. A similar approach is used by Franz et. al. [16] arguing that the large objects with small number of hot fields may have alignment issues. A cold field which has to be loaded with a hot field just because they are located in the same block of memory mapped to one cache line, wastes cache space.

Another less popular solution is data reordering, which usually applies to large arrays and data structures. In data reordering techniques, data arrangement in the memory is optimized for the program using it. The efficiency of these techniques has been argued for programs executed purely on the CPU. Badawy et. al. [10] argue that prefetching policies outperform the locality optimization methods (including data reordering) when sufficient memory bandwidth is available. In a limited memory bandwidth condition, such as slow memory or highly parallel multi-core systems, the locality optimization can have the lead if implemented properly.

2.2 Data Reordering

Data reordering to improve cache performance is an old and well-known idea. McKee et. al. [29] show that a simple data access reordering can cause near optimal cache utilization

without complicated hardware modification or kernel optimization. Herrero et. al. [21] compare block data reordering with a hyper-matrix tree data structure for matrix multiplication and verified that the flat reordered data outperforms the complex data structure.

But data reordering remained an academic exercise with a small impact on real world applications. The main reason is the poor overall performance due to the complexity of the reordering process. The savings in memory latency are usually compensated by the extra CPU computations for address calculation. This argument is supported by Günther et. al. [20]. They optimized a PDE solver on a hierarchical data structure using space-filling curves. The tree data structure is replaced with a multi-grid data structure, and the grid ordering is optimized with space-filling curves to enhance cache utilization. They report that despite the substantial cache utilization improvements, they could not improve the running time. The hierarchical space filling curves they use are too complex to compete with simple row- or column- order indexing.

Data ordering cache performance depends on the data access pattern in the program. This ordering is sometimes trivial with known data access patterns, such as matrix multiplication and image convolution. However, if the data access pattern is spatially-coherent but unknown ahead of time, it becomes necessary to optimize the ordering for an average general case. An example of a program with spatially-coherent access pattern is the line-integral algorithm, which appears in computer tomography, volume ray tracing, and spatial data projection. During the line-integral algorithm, a random line is traversed through n-dimensional data and the data is accessed in sequence along the line. Although the access pattern is spatially coherent it changes for every line and the optimal ordering cannot be pre-determined.

A general metric that correlates with many of the spatially coherent data access patterns is the Euclidean distance. Closer data points are more probable to be used together within a short period of time. Based on this idea, two popular heuristics for data mapping are minimizing the distance distortion and maximizing the locality preservation. By keeping the neighboring data points in n-D close to each other in 1-D memory, the optimum mapping increases the spatial locality of the data points in the memory. It increases the chance of using the neighboring memory cells in the near future. The neighboring cells are either loaded in the cache because of being in the same cache line, or they might be prefetched by the prefetcher. On a larger scale, data points in the same page can be viewed as neighbors occupying the same TLB item and sharing the same disk to memory operation.

The minimum distance distortion mapping for 1-dimensional data is trivial—the location in the array can be used as the memory index. The problem arises when data is logically n-dimensional. For higher dimensions, the optimal ordering is less trivial since the number of neighbors for each data point increases. For 2-D arrays, the default layout in most compilers is the so-called row- or column-major layout (referred to as **Scanline order**) shown in Figure 2.1a. This is a consequence of the simple nested loop typically used for n-dimensional indices. Scanline order behaves poorly in terms of distance distortion, unless the data is accessed in the same scanline order. In order to decrease distance distortion, data is sometimes divided into equally sized blocks each of which are ordered in Scanline order. This ordering is called **Block order** which is shown in Figure 2.1b. A more complex solution is using hierarchical ordering with *space filling curves* introduced in the next section.

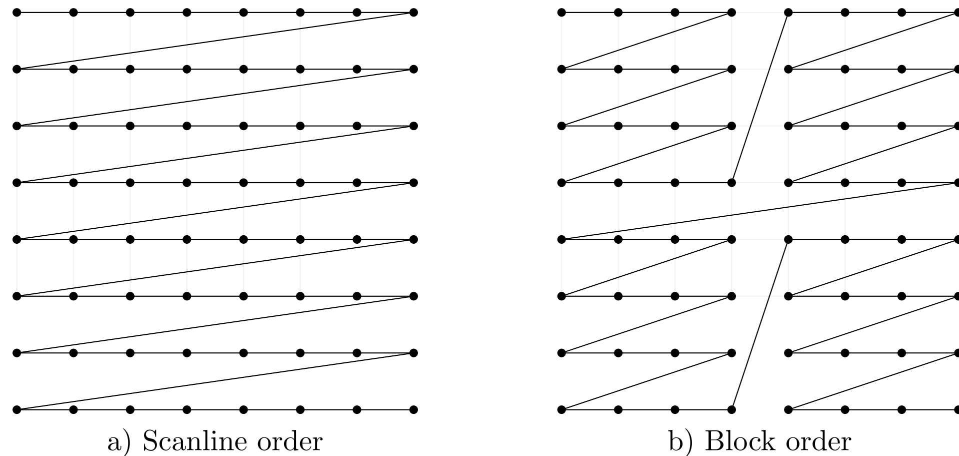


Figure 2.1: Data ordering in 2D for 8×8 data: a) Scanline order; b) Block order with a block size equal to 16 data points.

2.3 Space-Filling Curves and Morton Order

Space filling curves are continuous curves that pass through all n-dimensional data items. Since they were initially introduced by Giuseppe Peano in 2-D, 2-dimensional space filling curves are also called **Peano curves** [35, 5]. They are usually defined hierarchically in different resolutions. One sample case in 2-D is shown in Figure 2.2.

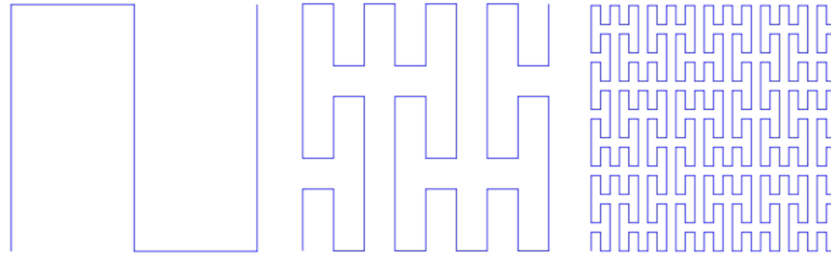
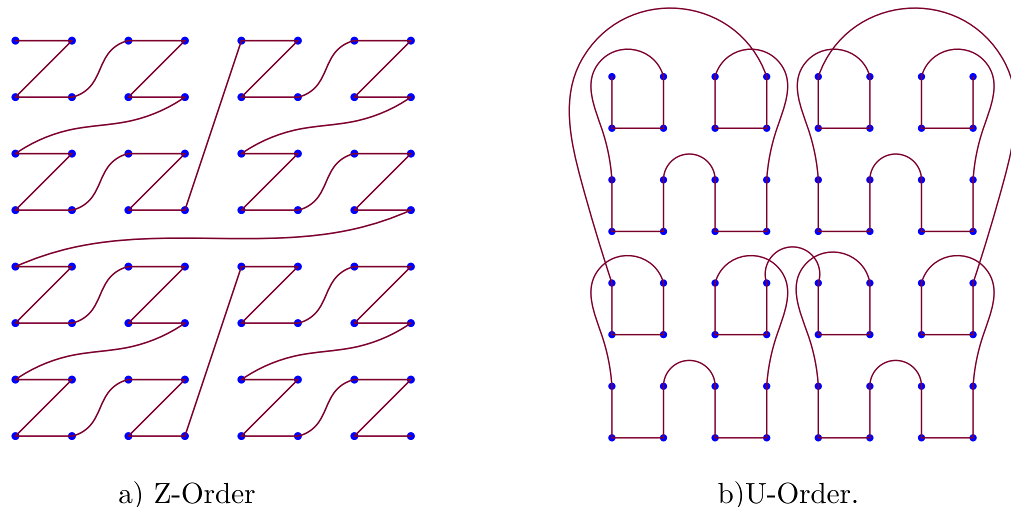


Figure 2.2: Three iterations of a Peano curve construction. [5]

There are a wide variety of space-filling curves with different properties. One of the simplest hierarchical space-filling curves is **Morton order** first defined by Morton in 1966 [30]. Morton order construction is done recursively and is shown in 2D in Figure 2.3. The iteration starts with a simple Scanline ordering of a unit n-dimensional cube with a single data point at each corner, which amounts to 2^n total data points. These points are then replaced with identical cubes recursively, which are themselves indexed in Morton order. Connecting these cubes together results in the global Morton order, as shown in Figure 2.3. Since the ordering inside each unit in 2D looks like the letter "Z", Morton order is also called **Z-Order** in the literature. By switching rows and columns, we get a similar ordering with similar properties called **N-Order**. The ordering inside the box can also be done in "⊔" shape, which is referred to **U-Order** in 2D.



a) Z-Order b)U-Order.

Figure 2.3: Z-order and U-order; different generations of Morton order.

Morton ordering is widely used in graphics applications due to its small distance distortion. Morton index calculation is also quite simple compared to other space-filling curves. Given the binary form of a n-dimensional index, the Morton index can be calculated by interleaving the bits into a single binary number. For example given a 2D index (X, Y) in a Cartesian square with $SIZE_X \times SIZE_Y$ points, the Morton index, M, can be calculated as:

$$X = (x_m x_{m-1} \cdots x_1 x_0)_2 \,, \tag{2.1}$$

$$Y = (y_m y_{m-1} \cdots y_1 y_0)_2 \,, \tag{2.2}$$

$$M = (y_m x_m y_{m-1} x_{m-1} \cdots y_1 x_1 y_0 x_o)_2 \,, \tag{2.3}$$

while the Scanline index (row-order), S, is:

$$S = X + Y * SIZE_X. \tag{2.4}$$

The Morton index calculation is computationally more expensive but has some benefits over the Scanline index. Beside less distance distortion, the Morton index is independent of the data dimensions of the Cartesian lattice.

Morton ordering is used in a wide range of applications to optimize data access latency on CPU and GPU. Knittel [23] used Morton ordering together with data padding to allow real-time CPU based volume rendering on Pentium-III processors, which had been a great success at the time. Nocentino [32] used Morton ordering to reduce memory access transactions on the GPU. Lauterbach et. al. [25] took advantage of Morton ordering in building bounding volume hierarchies quickly and efficiently on the GPU. Connor and Kumar [14] used Morton ordering to construct k-nearest neighbor graphs.

Morton ordering is also widely used in matrix operations. Athanasaki and Koziris [9, 8] optimized cache misses for matrix multiplication by using different combinations of Z-Order and N-Order for different levels of hierarchy. ElGindy and Ferizis [15] verified the effectiveness of Morton ordering in Strassen's matrix multiplication algorithm.

Compiler support for the Morton index has been proposed by David S. Wise and K. Patrik Lorton. Wise et. al. [41] claimed that Morton order on a C-to-C translator can achieve 67% performance improvement. They also introduced an optimized compiler called OPIE [17] that transforms C code written for row-major matrices into equivalent code for Morton order matrix representation. Morton order in different levels together with padding and loop unrolling are widely used by Wise [40] and Lorton and Wise [27] for C and

FORTRAN based matrix multiplication. Their reports on cache improvement is convincing but their time improvements are inconsistent and have been questioned since [38].

There are several other space-filling curves introduced based on Morton ordering. Some popular ones are **Morton-Gray order**, **Gray-Morton order** and **Double-Gray Morton order**. These orderings are produced by combining Morton bit-interleaving with Gray coding of the indices. Double-Gray Morton order is theoretically the most optimal space-filling curve in terms of distance distortion, but it is not widely used due to the complexity of its calculation. A 2D example of Double-Gray Morton order is shown in Figure 2.4.

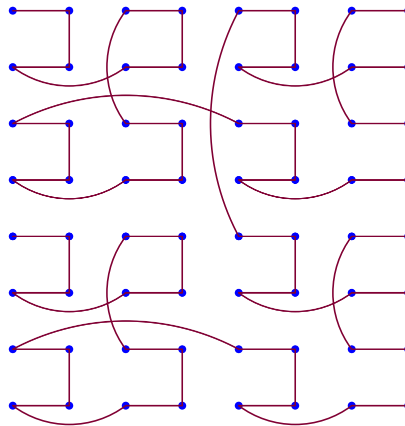


Figure 2.4: Double-Gray Morton: Optimal locality distortion ordering based on Morton order. The ordering in each 2×2 cell is reversed to reduce the average distance. The same pattern repeats in higher levels.

2.4 Index Calculation

Although the Morton index looks simple to calculate mathematically, it is not trivial on a typical processor. Fast bit interleaving is not supported for most of the processors, which makes the Morton index calculation a challenge. The most popular Morton index calculation method is based on the concept of *dilated integers* introduced by Wise [40]. The index calculation is done by converting the n-D indices to dilated integers by adding zero bits in between [34], and combining them by `SHIFT` and `OR` operations to make the Morton index. Adams and Wise [6] introduced arithmetic operations on dilated integers as a tool to speed-up the index calculation of the neighboring samples. These index calculation methods are generally used when Morton order is applied to data. But the performance of the Morton

order with this index calculation is questioned. Thiyagalingam et. al. [37, 38] believe the performance difference between simple row-ordering and simple column-ordering in 2D can be as large as a factor of 10, while Morton ordering's typical implementation is only slightly better than the worst of the two. They improve Morton order (sometimes close to the better of the two simple scanline orders) by taking advantage of loop unrolling, data alignment, and using a look-up table for address calculation. All of their results, however, were obtained by simulated profiling rather than hardware monitoring with performance counters.

2.4.1 Morton Index with Dilated Integers

The Morton index can be seen as n 1-D indices interleaved into one number. This is different from the Scanline index which steps in the next dimension after finishing one. The most popular way of calculating the Morton index is interleaving each component with a series of zero bits, and then combining all with proper `SHIFT` and `OR` operations.

In the remainder of this section we introduce the algorithm and its proof. For simplicity, we define the **log-base-b** form of integers, which represents the integer with base-2^b, and every digit can be represented with b binary bits.

Definition 2.4.1. ***log-base-b Number*** *An integer number X in Logbase-b is shown as:*

$$[A_m \cdots A_2 A_1 A_0]_{[b]} \sim [A_m \cdots A_2 A_1 A_0]_{2^b} \tag{2.5}$$

where $0 \leq A_i < 2^b$ for all $i \in \{0, 1, \cdots\}$ and $X = \sum_{i=0}^{m} (2^b)^i A_i$. The binary form of X is Base-2 or LogBase-1:

$$[x_m \cdots x_2 x_1 x_0]_{[1]} \sim [x_m \cdots x_2 x_1 x_0]_2 \tag{2.6}$$

Definition 2.4.2. ***n-Dilated Form*** *The n-dilated form of an integer X with binary form $[x_m x_{m-1} \cdots x_1 x_0]_2$ is $\overline{X}^{(n)}$ defined as:*

$$\overline{X}^{(n)} = \left[x_m \underbrace{0 \cdots 0}_{n-1} x_{m-1} \cdots \underbrace{0 \cdots 0}_{n-1} x_1 \underbrace{0 \cdots 0}_{n-1} x_0 \right]_2 = \sum_{i=0}^{m} 2^{ni} x_i \tag{2.7}$$

Suppose we want to calculate the Morton index M for a 2D point (X, Y) from the

2-dilated form of the 2D indices:

$$\begin{aligned}
X &= [x_m x_{m-1} \cdots x_1 x_0]_2 = \sum_{i=0}^{m} 2^i x_i \\
Y &= [y_m y_{m-1} \cdots y_1 y_0]_2 = \sum_{i=0}^{m} 2^i y_i \\
\overline{X}^{(2)} &= [x_m 0 x_{m-1} 0 \cdots 0 x_1 0 x_0]_2 = \sum_{i=0}^{m} 2^{2i} x_i \\
\overline{Y}^{(2)} &= [y_m 0 y_{m-1} 0 \cdots 0 y_1 0 y_0]_2 = \sum_{i=0}^{m} 2^{2i} y_i \\
M &= \overline{X}^{(2)} \vee (\overline{Y}^{(2)} << 1) = \sum_{i=0}^{m} (2^{2i} x_i + 2^1 (2^{2i} y_i))
\end{aligned}$$

In a similar scenario for a 3D point (X, Y, Z), the Morton index M can be calculated as:

$$\begin{aligned}
X &= [x_m x_{m-1} \cdots x_1 x_0]_2 = \sum_{i=0}^{m} 2^i x_i \\
Y &= [y_m y_{m-1} \cdots y_1 y_0]_2 = \sum_{i=0}^{m} 2^i y_i \\
Z &= [z_m z_{m-1} \cdots z_1 z_0]_2 = \sum_{i=0}^{m} 2^i z_i \\
\overline{X}^{(3)} &= [x_m 00 x_{m-1} 00 \cdots 00 x_1 00 x_0]_2 = \sum_{i=0}^{m} 2^{3i} x_i \\
\overline{Y}^{(3)} &= [y_m 00 y_{m-1} 00 \cdots 00 y_1 00 y_0]_2 = \sum_{i=0}^{m} 2^{3i} y_i \\
\overline{Z}^{(3)} &= [z_m 00 z_{m-1} 00 \cdots 00 z_1 00 z_0]_2 = \sum_{i=0}^{m} 2^{3i} z_i \\
M &= \overline{X}^{(3)} \vee (\overline{Y}^{(3)} << 1) \vee (\overline{Z}^{(3)} << 2) = \sum_{i=0}^{m} (2^0 (2^{3i} x_i) + 2^1 (2^{3i} y_i) + 2^2 (2^{3i} z_i))
\end{aligned}$$

Since there is no instruction to do the dilation in most of the modern processors, an efficient way of calculating it is needed. One of the most efficient algorithms to do so with logical operations is defined as follows.

Definition 2.4.3. ***Bit Sequence*** *In any log base b (base 2^b), the minimum (zero) and maximum ($2^b - 1$) digit are shown by 0 and* ***1*** *respectively. In other words,* ***1*** *in binary form is b one bits.*

Definition 2.4.4. ***n*-Dilated Integer in log-base** *b: An integer number X is called n-Dilated in log-base b iff it can be written in the form:*

$$X = \left[\cdots A_2 \underbrace{0\cdots0}_{n-1} A_1 \underbrace{0\cdots0}_{n-1} A_0 \right]_{[b]} \tag{2.8}$$

Lemma 2.4.1. *if X is n-dilated in log-base 2^b, then X" defined as:*

$$X" = \left\{ X \vee \left(X << \left(2^{b-1}(n-1) \right) \right) \right\} \wedge \left[\cdots \boldsymbol{1} \underbrace{0\cdots0}_{n-1} \boldsymbol{1} \underbrace{0\cdots0}_{n-1} \boldsymbol{1} \right]_{[2^{b-1}]} \tag{2.9}$$

is n-diluted in log-base 2^{b-1}.

Proof.

$$\begin{aligned}
X &= \left[\cdots A_2 \underbrace{0\cdots0}_{n-1} A_1 \underbrace{0\cdots0}_{n-1} A_0 \right]_{[2^b]} \\
&= \left[\cdots B_5 B_4 \underbrace{0\cdots0}_{2n-2} B_3 B_2 \underbrace{0\cdots0}_{2n-2} B_1 B_0 \right]_{[2^{b-1}]} \\
X' = X << \left(2^{b-1}(n-1) \right) &= \left[\cdots B_5 B_4 \underbrace{0\cdots0}_{2n-2} B_3 B_2 \underbrace{0\cdots0}_{2n-2} B_1 B_0 \underbrace{0\cdots0}_{n-2} 0 \right]_{[2^{b-1}]}
\end{aligned}$$

$$\begin{array}{lccccccc}
X: & \cdots & 0\underbrace{0\cdots0}_{n-2}0 & \underbrace{0\cdots0}_{n-2}B_3 & B_2 & 0\underbrace{0\cdots0}_{n-2}0 & \underbrace{0\cdots0}_{n-2}B_1 & B_0 \\
X': & \cdots & 0\underbrace{0\cdots0}_{n-2}B_3 & B_2\underbrace{0\cdots0}_{n-2} & 0 & 0\underbrace{0\cdots0}_{n-2}B_1 & B_0\underbrace{0\cdots0}_{n-2} & 0 \\
\hline
X \vee X': & \cdots & \underbrace{?\cdots?}_{n-1}B_3 & \underbrace{?\cdots?}_{n-1} & B_2 & \underbrace{?\cdots?}_{n-1}B_1 & \underbrace{?\cdots?}_{n-1} & B_0 \\
M & \cdots & \underbrace{0\cdots0}_{n-1}\mathbf{1} & \underbrace{0\cdots0}_{n-1} & \mathbf{1} & \underbrace{0\cdots0}_{n-1}\mathbf{1} & \underbrace{0\cdots0}_{n-1} & \mathbf{1} \\
\hline
X" = (X \vee X') \wedge M: & \cdots & \underbrace{0\cdots0}_{n-1}B_3 & \underbrace{0\cdots0}_{n-1} & B_2 & \underbrace{0\cdots0}_{n-1}B_1 & \underbrace{0\cdots0}_{n-1} & B_0
\end{array}$$

$\square$

Lemma 2.4.2. *if* $b = \lceil \log_2 \lfloor 1 + \log_2(X) \rfloor \rceil$ *then* X *is* n*-diluted in log-base* 2^b *for every* $n \in \mathbb{N}$.

Proof. We know that X in binary form has maximum of 2^b bits because:

$$2^b \geq 2^{\log_2 \lfloor 1+\log_2(X) \rfloor} = \lfloor 1 + \log_2(X) \rfloor \geq \lceil \log_2(X) \rceil \geq \log_2(X) \tag{2.10}$$

So, X in log-base 2^b is just one digit, which has the condition to be n-dilated in log-base 2^b for every $n \in \mathbb{N}$. □

Algorithm 1 Dilating integer number X with n bits for n-D Morton calculation.

1: `CONSTANT` n=Number of Dimensions.
2: **procedure** DILATE(X) ▷ X is the number
3: $b = \lceil \log_2 \lfloor 1 + \log_2(X) \rfloor \rceil$ ▷ Smallest log-base where X is n-dilated
4: **while** $b > 0$ **do**
5: $X = \left\{ X \vee \left(X << \left(2^{b-1}(n-1) \right) \right) \right\} \wedge \left[\cdots \mathbf{1} \underbrace{0 \cdots 0}_{n-1} \mathbf{1} \underbrace{0 \cdots 0}_{n-1} \mathbf{1} \right]_{[2^{(b-1)}]}$ ▷ Eq. (2.9)
6: $b = b - 1$
7: **end while**
8: **end procedure**

Theorem 2.4.3. *The Algorithm 1 generates the dilated form of* X *with* $3 \lceil \log_2 B \rceil$ *operations, where* B *is the number of bits required to store* X.

Proof. From Lemma 2.4.2, X has the initial condition to be n-diluted in log-base-2^b. After each iteration, the base gets logarithmically smaller and the algorithm terminates at base-2^1. So, the final X is n-diluted at base-2.

Each iteration is one `SHIFT`, one `AND`, and one `OR` operation. And we have $\lceil \log_2 \lfloor 1 + \log_2(X) \rfloor \rceil$ iterations. Since B us equal to $\lfloor 1 + \log_2(X) \rfloor$, the total number of operations is $3 \lceil \log_2 B \rceil$. □

If the size of the index is fixed and known a priori, the loop in Algorithm 1 can be unrolled to reduce the overhead. Suppose the array index addressing is 32 bits. Then the maximum dimension size is 2^{32}. So, dilation for each dimension takes at most 3×5 operations. Specifically, for an n-dimensional unit volume, each dimension is limited by $2^{\lfloor \frac{32}{n} \rfloor}$, leading to $3 \times \lceil \log_2 \frac{32}{n} \rceil$ operations, or 12 for a 2D cube.

For a 2D cube limited to a 32-bit index, the dimension size is less than or equal to 16 bits, leading to an initial b of 4. So, the dilation algorithm is:

$$\begin{aligned}
X &\leftarrow (X|(X << 8)) \ \& \ [00-FF-00-FF]_{16}; \quad \{b=4\} \\
X &\leftarrow (X|(X << 4)) \ \& \ \underbrace{[0F-0F-0F-0F]_{16}}_{[\cdots\ 0000\ 1111]_2}; \quad \{b=3\} \\
X &\leftarrow (X|(X << 2)) \ \& \ \underbrace{[33-33-33-33]_{16}}_{[\cdots\ 0011\ 0011]_2}; \quad \{b=2\} \\
X &\leftarrow (X|(X << 1)) \ \& \ \underbrace{[55-55-55-55]_{16}}_{[\cdots\ 0101\ 0101]_2}; \quad \{b=1\}
\end{aligned} \tag{2.11}$$

And similarly, for a 3-D cube where dimension size is less than 11 bits, the initial b is again 4. Therefore, the dilation algorithm is:

$$\begin{aligned}
X &\leftarrow (X|(X << 16)) \ \& \ \underbrace{[FF-00-00-FF]_{16}}; \quad \{b=4\} \\
X &\leftarrow (X|(X << 8)) \ \& \ \underbrace{[0F-00-F0-0F]_{16}}_{[\cdots\ 0000\ 0000\ 1111]_2}; \quad \{b=3\} \\
X &\leftarrow (X|(X << 4)) \ \& \ \underbrace{[C3-0C-30-C3]_{16}}_{[\cdots\ 0000\ 1100\ 0011]_2}; \quad \{b=2\} \\
X &\leftarrow (X|(X << 2)) \ \& \ \underbrace{[49-24-92-49]_{16}}_{[\cdots\ 0010\ 0100\ 1001]_2}; \quad \{b=1\}
\end{aligned} \tag{2.12}$$

And for a 4-D cube, where the dimension size is less than or equal to 8 bits, the initial b is 3:

$$\begin{aligned}
X &\leftarrow (X|(X << 12)) \ \& \ \underbrace{[00-0F-00-0F]_{16}}_{[\cdots\ 0000\ 1111]_2}; \quad \{b=3\} \\
X &\leftarrow (X|(X << 6)) \ \& \ \underbrace{[03-03-03-03]_{16}}_{[\cdots\ 0000\ 0011]_2}; \quad \{b=2\} \\
X &\leftarrow (X|(X << 3)) \ \& \ \underbrace{[11-11-11-11]_{16}}_{[\cdots\ 0001\ 0001]_2}; \quad \{b=1\}
\end{aligned} \tag{2.13}$$

Once all the dimension indices are in n-dilated form, they can be combined to get the Morton index:

$$M = \overline{X}_0^{(n)} \vee (\overline{X}_1^{(n)} << 1) \vee (\overline{X}_2^{(n)} << 2) \vee \cdots \tag{2.14}$$

Since each dimension adds one `SHIFT` and one `OR` operation, it requires a total of$2(n-1)$ operations to combine the dilated dimensions. Hence, Morton index calculation with dilated integers in n-dimensions, where each dimension is less than B bits, requires $2n-2+3\lceil \log_2 B \rceil$ operations.

2.4.2 Morton Index with Look-up Tables

The Morton index calculation has a lot of computational overhead. A look-up table can be used to replace some computational parts. The look-up table should fit in the cache for fast index calculation. The more calculations are replaced with a look-up table, the more cache space is wasted on the table. The optimal trade-off depends on the cache space and processor to memory speed ratio.

The look-up table can be used in several different forms. In calculating the index with dilated integers, the dilation can be performed with a look-up table when the dimension sizes are reasonably small. For an n-dimensional cube, the size of the dilation look-up table is the nth root of the data size, and it should be accessed n times. For low-dimensional data the space overhead is quite large and it does not make sense to use such relatively large look-up tables. On the other hand, for high-dimensional data multiple table look-ups reduces the performance.

Another way of using look-up tables with more control on the trade-off is a pre-calculation of Morton indices for one fixed small block size. Since Morton is a hierarchical indexing, the global index can be calculated by several accesses of the look-up table, one for each hierarchical level. For data of size N, and a lookup table of size s, it will require $t = \log_s N$ table lookups. A smaller look-up table saves cache space but requires more table accesses and more computation to merge the levels.

2.5 Morton Address Update

By exploiting the fact that most of the applications only need to access the direct spatial neighbors of a data item, we can define a function which updates the index for the neighboring points, instead of calculating it from scratch. We do this by defining a special add function which adds two Morton indices together.

Consider an n-D data point, $(P_1, P_2, \cdots, P_n)$ with its associated Morton index, M_P. Suppose we would like to move to a neighboring data point, with offset vector $(D_1, D_2, \cdots, D_n)$, which would get Morton index M_D if it is seen as a point location by itself. The Morton address update technique calculates the Morton index of the new location $Q = P + D$, taking M_P and M_D as inputs. We call this operation **SpecialAdd** since it is some kind of a special addition function for Morton indices.

The Morton index update idea was introduced by Adams and Wise [6]. The technique

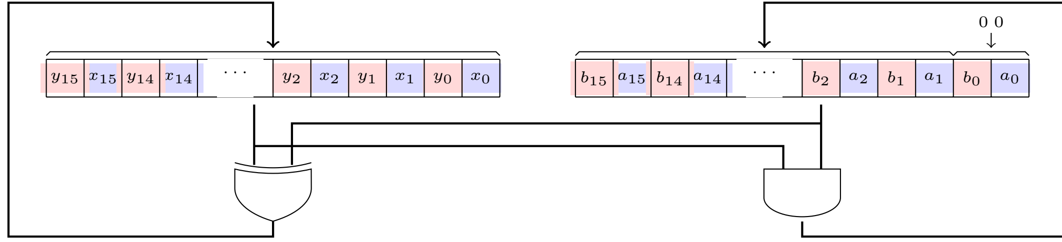


Figure 2.5: SpecialAdd function is implemented by modifying a serial adder to have n shifts for each carry bit instead of one shift.

they use is splitting Morton indices into dilated integers, one for each dimension. They developed arithmetic operations for dilated integers, with which they can calculate the position of the new point with dilated integers. The results are then merged to one Morton index for the new point, Q.

Our SpecialAdd function works differently. We do the index calculation directly without splitting it to dilated integers. Our method is based on implementing the add function using logical operations and forward the carry bit manually, as shown in Figure 2.5. The add function is implemented one bit at a time in each dimension. Since there is no interaction between bits from different dimensions, the logical `XOR` and `AND` operations are done in parallel on all dimensions. Theoretically, the carry does not propagate more than 2 bits on average, where the displacement index M_D is pointing to the direct neighbors. This makes the SpecialAdd function theoretically fast on average, specially in higher dimensions. Our implementation for the SpecialAdd function is shown in Algorithm 2.

Algorithm 2 Morton index update with index specialAdd.

`CONSTANT` n=Number of Dimensions.
procedure SpAdd(M_P, M_D) ▷ M_P and M_D are two Morton indices in n-dimensions
  $M_Q \leftarrow M_P$
  while $M_D \neq 0$ **do**
    $M_Q^{old} \leftarrow M_Q$
    $M_Q \leftarrow M_Q^{old} \oplus M_D$
    $M_D \leftarrow \left(M_Q^{old} \wedge M_D\right) << n$
  end while
    ▷ M_Q is Morton index for the sum of the two vectors with indices M_P and M_D
return M_Q
end procedure

The index update can also take advantage of a look-up table in systems with relatively large first-level cache. The idea is to update the first n-bit of the index with a table look-up if the carry does not propagate for more than n bits. Otherwise we calculate the index from scratch. Since the look-up optimization idea does not give any performance improvement with GCC optimization, we removed it from our simulations.

2.6 Hybrid Solution

The spatial and temporal locality has a limited effective range. In our line-integral application, we know that neighboring points inside the interpolation kernel will be accessed with probability 1. The access of points outside the interpolation kernel depends on the line direction and have lower probability to be accessed. With increasing distance, this probability decreases quickly. After a certain distance threshold, the access probability is very low and can be assumed as constant over the whole field. Data locality after this threshold does not have any noticeable effect on the performance.

Another factor that limits the importance of data locality in a larger neighborhood are the cache parameters. Neighboring data within the size of a cache line are loaded together. This gives a special importance on data locality in a neighborhood with the size of a cache line. Another important neighborhood size is the page size. Each page is given a row in address translation look-up and is also loaded at once to the main memory.

We claim that data locality optimization for every range and all hierarchical levels is not required for cache optimization. We propose two other indexing methods and compare them with Morton indexing in order to prove this claim.

2.6.1 Hybrid Morton

In hybrid Morton ordering, data is divided into equally sized blocks. The ordering inside each block is Morton ordering, while blocks are ordered in Scanline. This gives us data locality inside each block, with fast index calculation among different blocks. By picking the block size slightly larger than a page size, we guarantee locality at all smaller levels with Morton ordering. It optimizes cache line utilization as well as TLB and page loads without unnecessary Morton ordering for larger block sizes. To reduce the computational cost of Morton index calculation, the intra-block indexing is done by a table look-up that fits in L1 cache.

2.6.2 Dimension Shuffle Multi Block

This ordering is a more refined optimization by separately optimizing the locality for specific cache sizes. The data is hierarchically split into blocks in 3 levels. The 1st level of small blocks provides data locality for cache lines while the 2nd level of larger blocks guarantees locality for data pages. Since index calculation is costly in multi-block ordering, we use a different Scanline ordering at each level to reduce the blocking overhead. In 2D, it means switching between row- and column-ordering at each level. Figures 2.6, 2.7 and 2.8 show the position of each location bit in the final index in 2D, 3D and 4D, with regular 3-level blocking and dimension shuffle blocking.

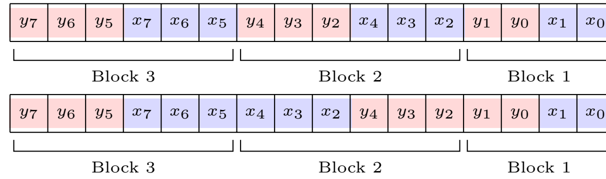


Figure 2.6: 3-Level block indexing for point (X,Y) in a 2D array of size 256 × 256: Top: Regular Scanline; Bottom: Dimension Shuffle.

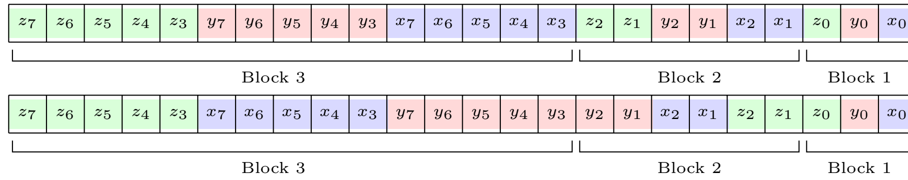


Figure 2.7: 3-Level block indexing for point (X,Y,Z) in a 3D array of size 256 × 256 × 256: Top: Regular Scanline; Bottom: Dimension Shuffle.

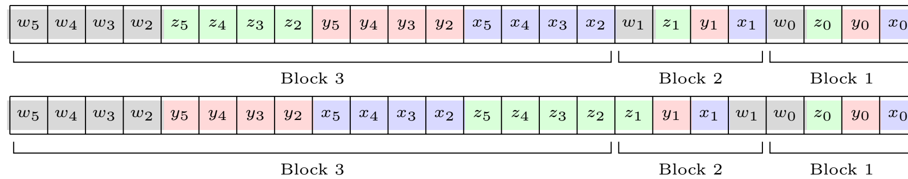


Figure 2.8: 3-Level block indexing for point (X,Y,Z,W) in a 4D array of size 64×64×64×64: Top: Regular Scanline; Bottom: Dimension Shuffle.

The algorithm to calculate the dimension shuffle blocking index is similar to the normal multi-block index algorithm with fewer number of bit segments. We also use index-update to speed up the index calculation for the neighboring data points. The index update is based on walking one step in only one dimension, as shown in Algorithm 3. The constant $MASK(d)$ is the mask to separate the bits from the target dimension in the index. The carry for the dimension increment in one block is carried forward to the next block by the complement of the mask. An example of the index increment procedure on the X dimension in 2D is shown in Figure 2.9.

Algorithm 3 Updating the dimension-shuffle index.

1: n=number of dimensions
2: ▷ S: old index, d: dimension to move, $dir = \pm 1$: direction
3: **procedure** DimShuffle_INC(S,d,dir)
4: $targetDimBits \leftarrow S \wedge MASK(d)$ ▷ Seperating bits
5: $targetDimBits \leftarrow targetDimBits + dir \times (\neg MASK(d) + 1)$ ▷ Adding a constant
6: $targetDimBits \leftarrow targetDimBits \wedge MASK(d)$ ▷ Masking invalid bits
7: **return** $targetDimBits \vee (S \wedge \neg MASK(d))$ ▷ Adding maked bits
8: **end procedure**

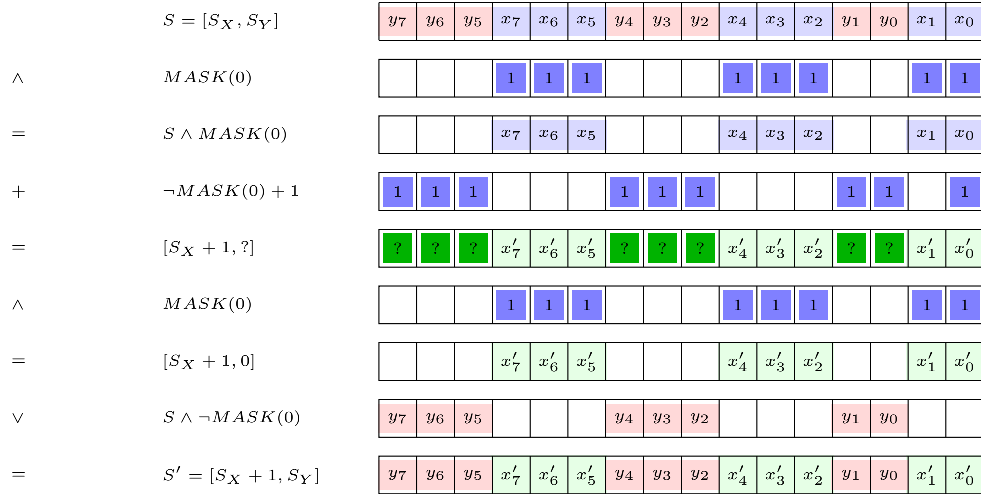


Figure 2.9: Index increment on the first dimension for 2D arrays. S is the initial dimension-shuffle index for the point (S_X, S_Y) and S' is the updated index for the point $(S_X + 1, S_Y)$.

2.7 Chapter Summary

N-dimensional data have spatial locality if the data points close to each other in memory are also close in n-dimensional space. Double-Gray Morton ordering is the most optimal ordering for spatial data locality but the index calculation is computationally expensive. Morton ordering is a near-optimal solution that has spatial locality in all distances, especially within blocks with size 2^i for every integer i. The Morton index can be calculated by interleaving the position of the point in each dimension. One efficient algorithm to do so is using dilated integers which is explained in Algorithm 1.

Due to data locality, we usually want to move to neighboring points. This can be done by updating the index instead of calculating it from scratch. We implement index update as a special add function for Morton indices, defined by bitwise logical operations in a loop, with 3 operations in the loop that is run twice on average. Despite significantly lower instruction count for this method compared to Morton index calculation with dilated integer, it is slower with GCC optimization since the loop is less predictable and cannot be unrolled.

We introduce a hybrid method which is a mixture of Morton and Scanline ordering. The whole space is divided into fixed sized blocks. Morton is used for ordering points inside each block where inter-block ordering is governed by Scanline ordering. The block size is chosen to be close to a page size.

Dimension shuffle block ordering is a hierarchical 3 level block ordering with Scanline ordering at each level. The order of dimensions for indexing each level is changed to reduce the number of segments of contiguous bits within one dimension (which we will refer to as a bit cluster) in the index. Bit clusters introduce overhead during the index calculations. The first block size is chosen with a size close to the cache line, while the second level is close to a page size.

The idea of fixed block indexing was introduced by Grimm et al. [19] that propose data reordering for GPU optimization. They had 2-level blocking with Scanline for inter- and intra-block ordering. They repeated their experiments with different block sizes and observed the existence of an optimal size of 64KB.

Chapter 3

Experiments

We designed a set of experiments to study the effect of data reordering defined here and in the literature. The effect of Morton ordering has been studied for years. Many papers confirm improved cache utilization and miss rate optimization although there are very limited time improvement reports. The different ordering methods we study in this section are:

- `Optimal`: It is a lower bound[1] for performance metrics. We just access the same single block of memory over and over as a representative of the whole n-dimensional array. Since one variable is representing the whole array, it stays in the cache or even CPU registers with minimal overhead;
- `Scanline`: The original row-order indexing used in C/C++ and many other compilers by default;
- `Morton`: Morton ordering with the index calculated by dilation, defined by Algorithm 1 and Eq. (2.14);
- `SpAdd`: Morton ordering with index update inside the filter kernel. One location index is calculated using dilation while the indices for other neighboring locations inside the filter kernel are calculated by the special add function defined by Algorithm 2;
- `LUT`: Hybrid Morton-Scanline indexing. The n-dimensional space is divided into equally fixed size blocks. The intra-block ordering is Morton while the inter-block

[1] Not a tight bound since it is not accessing correct data points

Table 3.1: The cache and memory properties for Intel SandyBridge i5-2300. [4, 3, 1, 2]

	Type	Assoc.	Size	Line Size	Latency	Bandwidth
L1 Data	Single Inclusive	8	32KB	64 Bytes	4 Clk	89.6GB/s
L2	Single Inclusive	8	256KB	64 Bytes	12 Clk	89.6GB/s
L3	Shared Non-Inc.	12	6MB	64 Bytes	26∼31 Clk	89.6GB/s
Memory	Shared	Full	6GB	–	N/A	21 GB/s

ordering is Scanline. The intra-block Morton index is calculated using a look-up table;

- `DimShuffle`: Dimension shuffle multi-block ordering. The space is divided into fixed sized blocks in several levels hierarchically. The ordering of the cells and blocks at each level is Scanline, but the dimension order is optimized (minimizing bit clusters) at each level.

3.1 Experiment Setting

As a case study we use the computation of a 1D line in an n-dimensional unit hyper-cube, where $n \in \{2, 3, 4\}$. Typical use cases of this algorithm are volume ray-tracing, computer tomography and data projection. A random line is selected by choosing two random points in the boundary of the unit hyper-cube. The hyper-cube is sampled on a Cartesian lattice with C^n samples (C samples in each dimension), stored in an n-dimensional array. The line integral is calculated using a Riemann sum over equidistant samples, with the distance equal to $\frac{1}{C}$, which is equal to the Nyquist rate for perfect reconstruction.

The simulations are done on an Intel Sandy Bridge Core i5-2300 at 2.80GHz maximum clock speed. The physical address size is 36 bits and the virtual address size is 48 bits. But we only use 32-bit integers as array indeces. Table 3.1 shows the properties of the different cache levels and main memory, with their theoretical latency and maximum bandwidth. The system was running Linux with kernel version 2.6.38.

We run several instances of the code as independent threads to measure the performance when more than one core is busy in the system. The profiling is done by measuring hardware performance counters using the Likwid 2.3.0 Profiling Tool [39]. Hardware performance events and metrics used in this project are listed in Appendix A.

3.2 Results

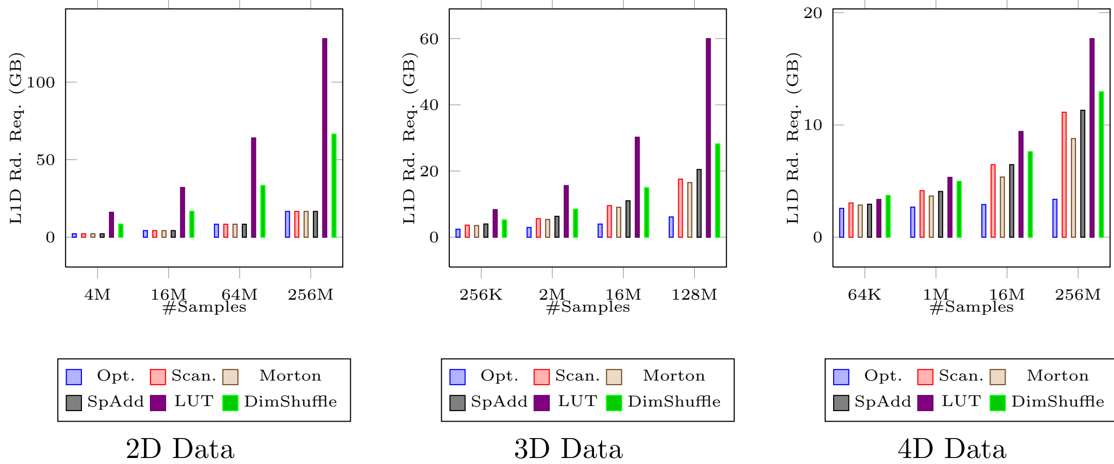


Figure 3.1: Average size of data (in GB) loaded on each CPU core; or equivalently, the data requested from the L1 cache of each core. While the graph represents measurements for four threads, there is no difference for 1, 2, 3, or 4 threads.

3.2.1 Data Load Requests

The average size of the loaded data per thread or, equivalently, the volume of requested data for each thread in a multi-threaded setting for different ordering schemes is shown in Figure 3.1. A load request can be a request for a data point, a temporary variable, a table look-up, or data structure properties such as sizes and data strides. The number of threads has no effect on the data request rate as expected due to the independence of the threads. So, we just show the results for a 4-thread simulation as a representative for all cases. The `Optimal` bars show the minimum data requests, which is accessing data point and temporary variables required for the line integral calculation. `Scanline` ordering has an extra overhead for index calculation, which is the request for data size or data stride at each dimension. This overhead increases in higher dimensions. The two implementations of Morton ordering, `Morton` and `SpAdd`, calculate the index with less dependence on the data size. The only extra data request overhead is the total data size, to check if the calculated index is inside the data block. The `LUT` method has the maximum data access overhead due to the table look-ups at each index calculation in addition to the data size at each dimension due to the Scanline ordering at the highest hierarchical level. The other block-based ordering

method, **DimShuffle**, also uses Scanline for inter-block ordering, which requires data size at each dimension. This method has an extra overhead at the beginning of the program, which is the pre-calculated shifts and masks for the accessed data size.

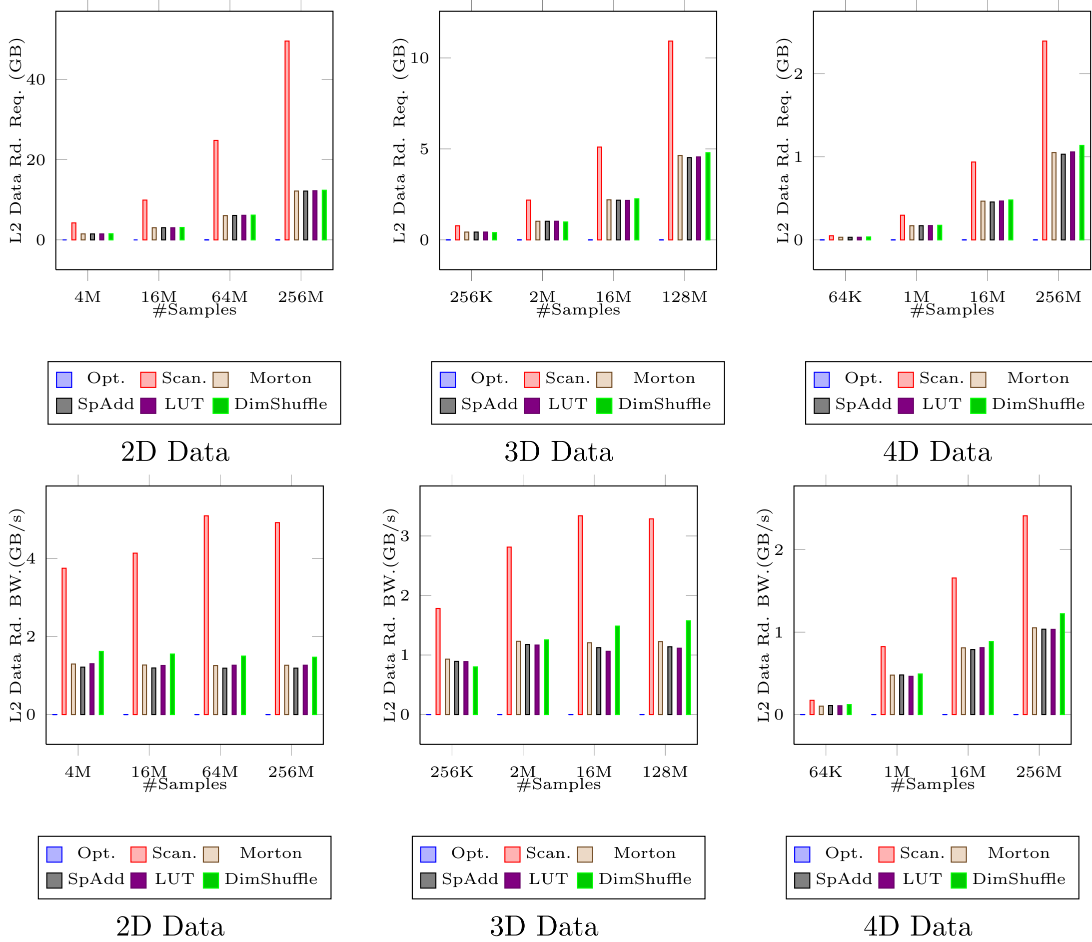


Figure 3.2: Average data transfers from L2 to L1 cache at each core; or equivalently, the data requested from L2 cache of each core, for 1/2/3/4-threaded line integral algorithm; First row: data size; Second row: average bandwidth.

3.2.2 On-Core Cache Utilization/Bandwidth

Figure 3.2 shows the average size and bandwidth of the data load requested from L2 cache, as a result of L1 data cache misses. Although we had similar, or in some cases higher data requests compared to *Scanline* on Morton and Block orderings, the number of cache misses at the L1 level is much smaller. Improved spatial locality for data points is the main reason

for lower cache misses at the L1 level, and subsequently, lower L2 data requests. The higher spatial locality of the data has two important effects on the execution behavior. Since a cache line is 16 data points in our case (16 floats each taking four bytes), each L2 request loads 16 data points in the L1 cache. In a low spatial locality case, as in *Scanline*, only a small portion is used and the rest might be removed from the cache before being used. In a case of high locality, it is more probable that the other loaded data points will be used shortly afterwards. This effect is more visible with the presence of a prefetcher, since a prefetcher relies on spatial locality. The other effect impacting the results is the reduced cache set conflicts for the neighboring data that might be used together. Data points matched to the same cache set are further from each other in Morton and Block ordering. This means the data mapped to the same cache set are not likely used together, and accessing one means that the other one has not been used in the near past, or will not be used in the near future.

Another reason for the huge difference in L1 and L2 accesses, especially in LUT indexing, is high temporal and spatial locality for temporary variables and look-up tables. Since these are the major overhead of LUT compared to *Morton*, the differences diminish in the L2 data accesses.

One of the most important observations for the L2 data access comparisons is the similarity of the results for Morton and two Block indexing methods introduced. Figure 3.2 confirms that the global optimization for data ordering is not mandatory to achieve high cache utilization. Block ordering can be as effective as Morton ordering for the proper block size, regardless of the ordering inside and outside each block.

3.2.3 Off-Core Data Access and Main Memory

A similar behavior to L2 cache data loads can be seen in L3 cache data loads. Figure 3.3 and Figure 3.4 show the total data requests in total volume and bandwidth from the L3 cache, which is shared among all cores. Since it is the result of on-core L2 cache misses, it is still independent of the number of active cores. These figures show flexibility on inter- and intra-block ordering even in the last level of cache. In such a scenario, it does not make sense to stick with Morton ordering, or any other complex ordering with expensive index calculations, just because it keeps block locality at all hierarchical levels. As we confirm here, the order of the blocks does not impact the spatial locality.

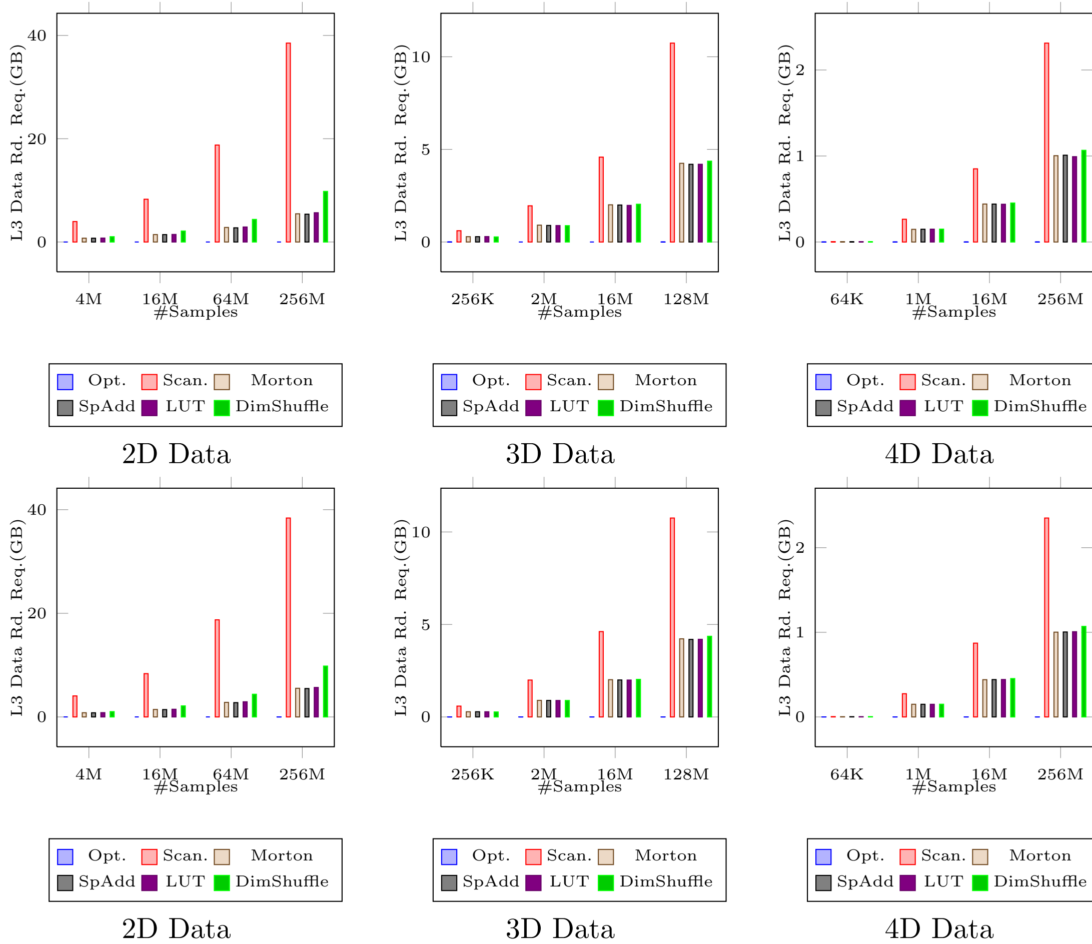


Figure 3.3: Average size of data transferred from L3 to L2 on each core; or equivalently, the data requested from L3 cache from each core. Top row: Single-threaded; Bottom row: quad-threaded.

Figure 3.5 shows the data size missed in L1 and LFB[2], which is then requested from the L2 cache averaged over all threads. Figure 3.6 shows the portion of these data requests covered by L2 and L3 cache levels. The first and second rows show the portion of data received from L2 hits and L3 hits respectively. The number of L2 and L3 data hits are relatively small compared to the total L2 data requests. It shows the inefficiency of the prefetcher and data ordering in taking full advantage of L2 and L3 caches.

The spatial locality principle says that for a certain application the data close to the

[2]Line Fill Buffer: when an L1 cache miss occurs and the data is located in a previously requested block, the request will be placed in the LFB to prevent duplicate accesses to L2 cache.

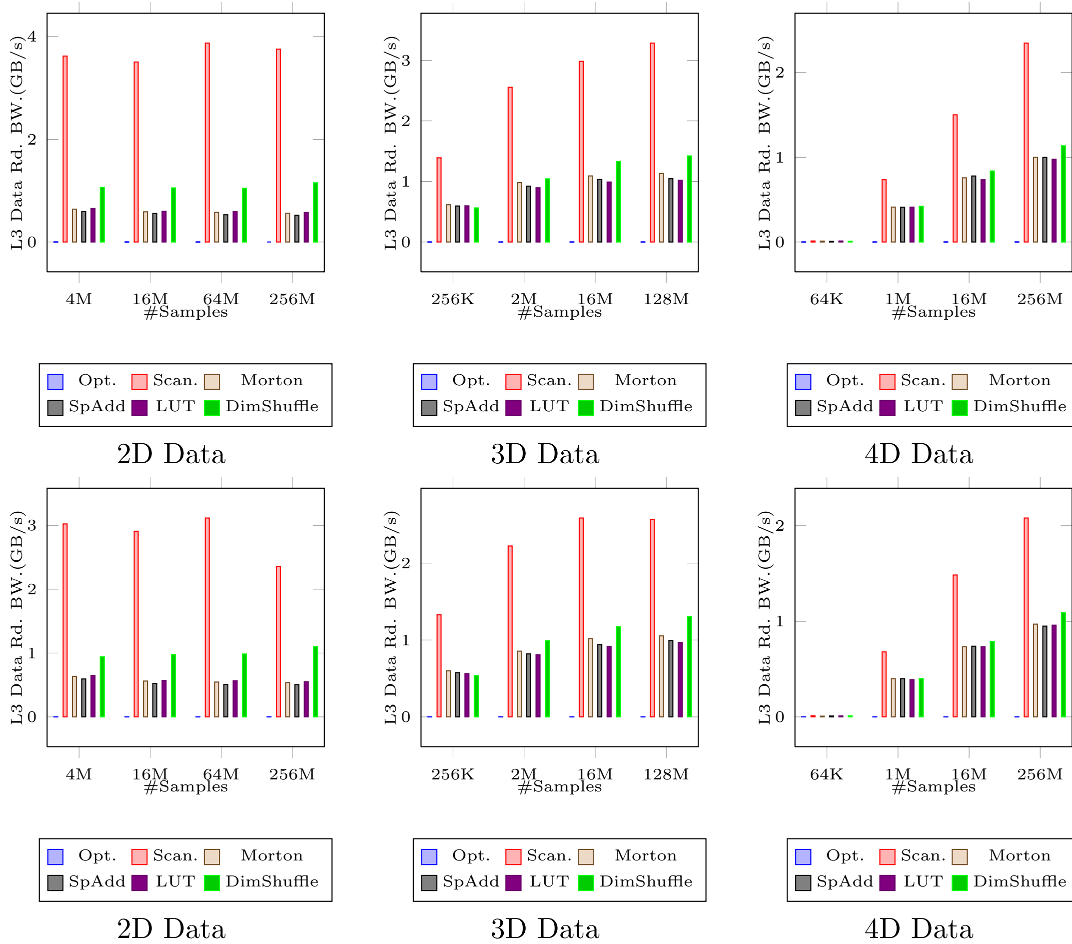


Figure 3.4: Average bandwidth used to transfer data from L3 to L2 for each core; or equivalently, the data bandwidth requested from L3 cache by each core. Top row: Single-threaded; Bottom row: quad-threaded.

currently accessed point has higher probability to be accessed next compared to the points further away. This locality feature has a maximum limit, depending on the running algorithm. For our line integral scenario, with each data point, we access direct neighbors with very high probability due to the interpolation kernel. It is also very possible to access the points with distance 2 and 3 while we are moving in the direction of the line. The locality beyond this space is not considered "locality" due to the low access probability, which can be just considered equal to the rest of the space. In our application scenario with one thread per core, the locality region completely fits inside the L2 cache. Any replacement policy for the L3 cache is equivalent to random data placement and has low performance. So,

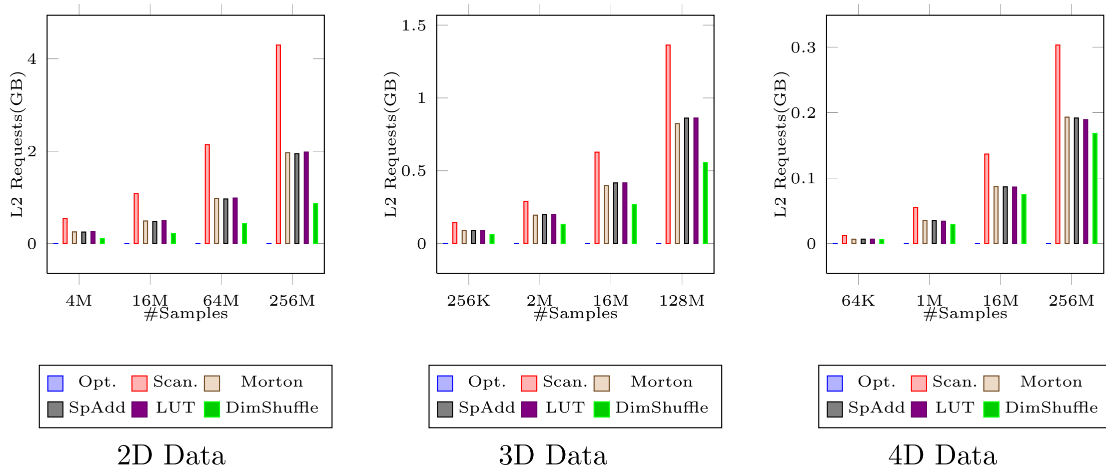


Figure 3.5: L2 cache requests: L1/LFB cache misses, which needs access to L2 and higher.

the existence of L3 cache has no benefit unless it can store most of the data, where even a random placement can result in a high hit rate.

Off-core cache level and main memory are shared resources among different cores and processors in a multi-core or multi-processor system. Clearly, more data accesses from main memory or relatively slow off-core cache slows down the code execution and wastes energy in the system. But beside the total value, the requested bandwidth from these shared resources has another inevitable effect on other processes on the system. Memory has a limited bandwidth that should be shared among all executing processes on all cores. If one process occupies a big portion of the bandwidth, other processes can also slow down due to congestion on a shared resource. This issue is more serious in low-bandwidth memory systems, external memories and swap memory, and data accesses over a network. We observe over 3 times more memory accesses per instruction with Scanline ordering over Morton and Block ordering for 256MB data in 2D. Although this might not be a problem in a single thread single processor setting, it can dramatically reduce the performance when memory bandwidth is shared among different processors.

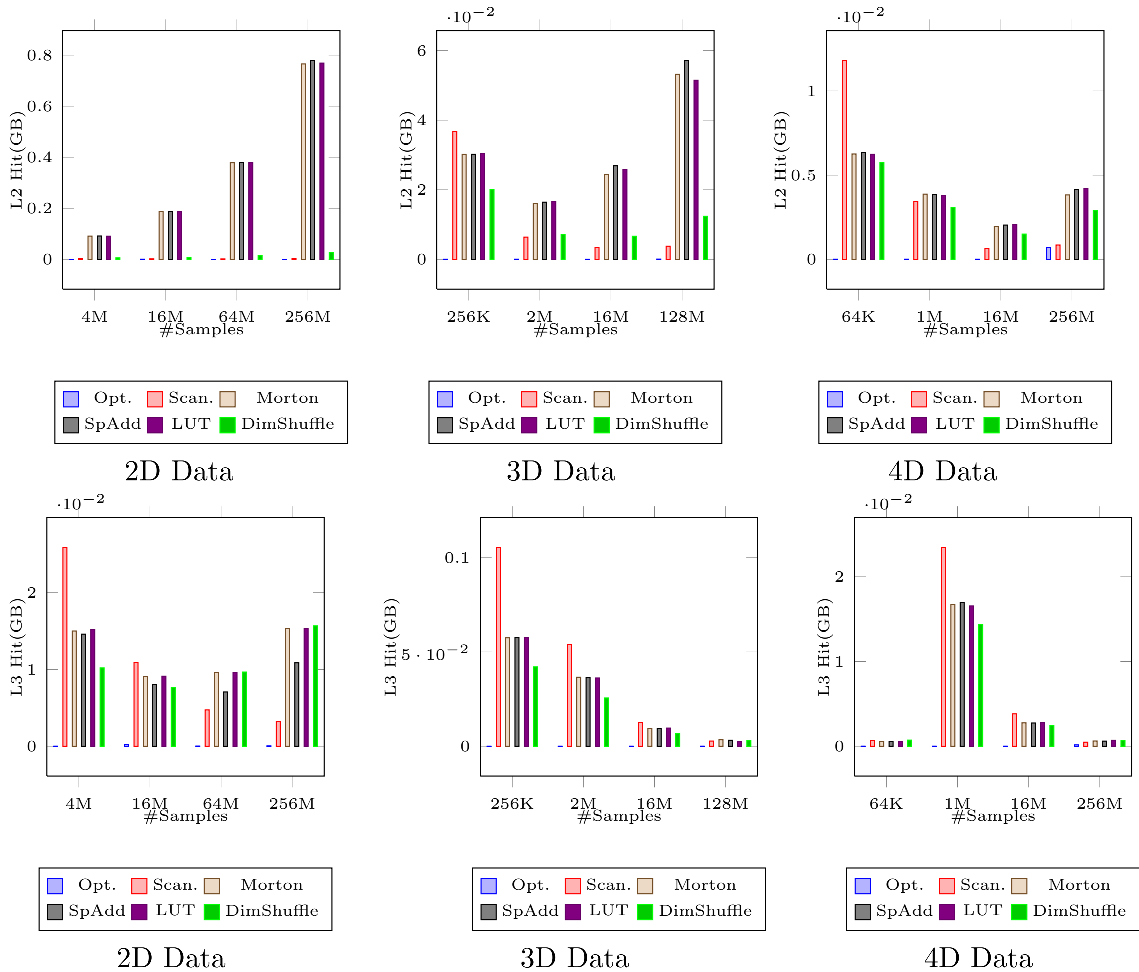


Figure 3.6: The performance of L2 and L3 caches: First row: Data loaded from L2 as source; Second Row: Data loaded from L3 as source.

3.2.4 TLB and Paging Effects

Virtual-to-Physical address translation is a necessary step for data loads in L1 cache misses. This translation is done using a look-up table partially cached in the L1 DTLB[3] cache. Similar to data cache misses, DTLB cache misses cause walking down the memory architecture to find the page table item. This process adds a large delay to data access since it requires an access to the main memory. Figure 3.7 shows the DTLB miss rates at the L1 cache level. In the 2D case, L1 DTLB misses for Morton and Block orderings are less

[3]Data TLB; Portion of the TLB assigned only for data address translation.

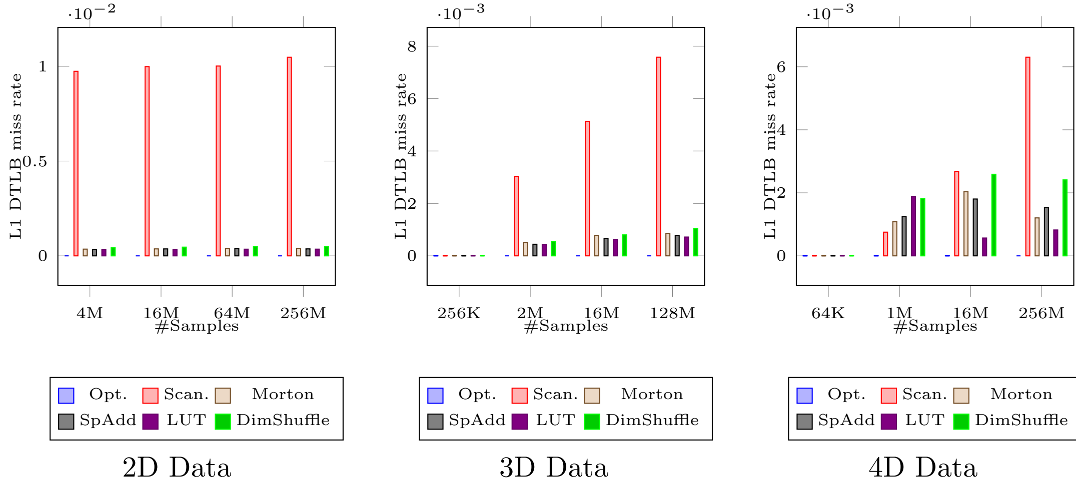


Figure 3.7: L1 DTLB miss rate (number of misses per instruction), for the single-threaded line integral algorithm.

than 0.04 times the misses for Scanline, which is a dramatic improvement. In the 3D and 4D cases fewer DTLB misses occur in Scanline, as a result of a smaller line stride. Block ordering on the other hand does not perform well due to the absence of a block with size 4KB, equal to the default page size. In the 2D case, a 32×32 block filled with floating point data completely fills a page, while the closest block size in 3D is $8 \times 8 \times 8$, or 2KB, and in 4D is $4 \times 4 \times 4 \times 4$ or 1KB. One solution to this problem is to treat dimensions differently and use a non-cubic block to make it exactly equal to a page size. Morton ordering can be seen as blocks with size exactly equal to 4KB. As expected we see more efficient TLB hit rate for Morton ordering compared to other orderings in 3D. This experiment confirms the importance of discrete block sizes in data ordering.

3.2.5 Overheads and Run-time

Figure 3.8 shows the total number of retired instructions with different indexing methods. The `Optimal` case shows the baseline, which is the actual running algorithm, without indexing overhead. `Scanline` has the minimum index calculation overhead since it is one multiplication and one addition per dimension. The dilated integer Morton index calculation, marked as `Morton` in the figure, has more than 57% overhead in 2D, 36% in 3D, and 13% in 4D, compared to `Scanline`. The index update optimization method, `SpAdd`, has not been very successful after GCC optimization and has more instructions than `Morton`,

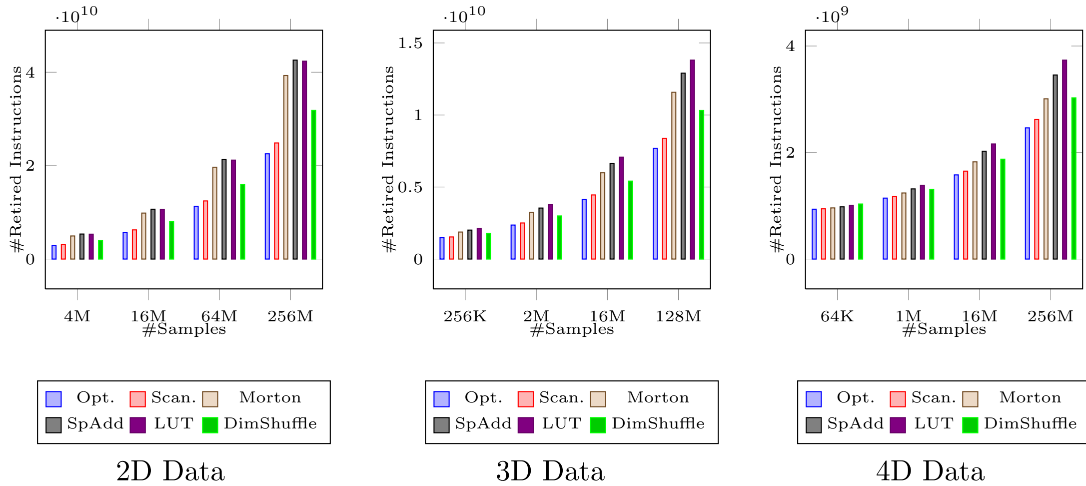


Figure 3.8: Total number of retired instructions. Single-threaded line integral.

especially in higher dimensions[4]. Among the two block based methods, the look-up based Morton-Scanline method, `LUT`, is not a good replacement for Morton indexing since it has the computation overhead slightly over `Morton` after GCC optimization. The dimension shuffle blocking method, `DimShuffle`, on the other hand has the best performance among the reordering algorithms with the overhead comparable to `Scanline`. Since the performance of the dimension-shuffle method in cache and TLB utilization is similar to Morton ordering, it is a good replacement for Morton ordering due to its low index calculation overhead.

Figure 3.9 shows the total run-time for each case. Since the calculation overhead is relatively high for `Morton`, `SpAdd`, and `LUT`, the time saved on memory accesses are wasted on running extra instructions and there is no time improvement in these cases. There are even some instances where the run-time increases despite less accesses to the high latency main memory. The most optimal block based method in terms of runtime is the new dimension-shuffling method due to its low computational overhead.

3.2.6 Energy Consumption

Figure 3.10 and 3.11 show the power and energy used by the processor to run the code in different cases. The power consumption for all cases is consistent around 13.5 ± 0.5

[4]Without GCC optimization, `SpAdd` performs slightly better than `Morton` in terms of retired instructions.

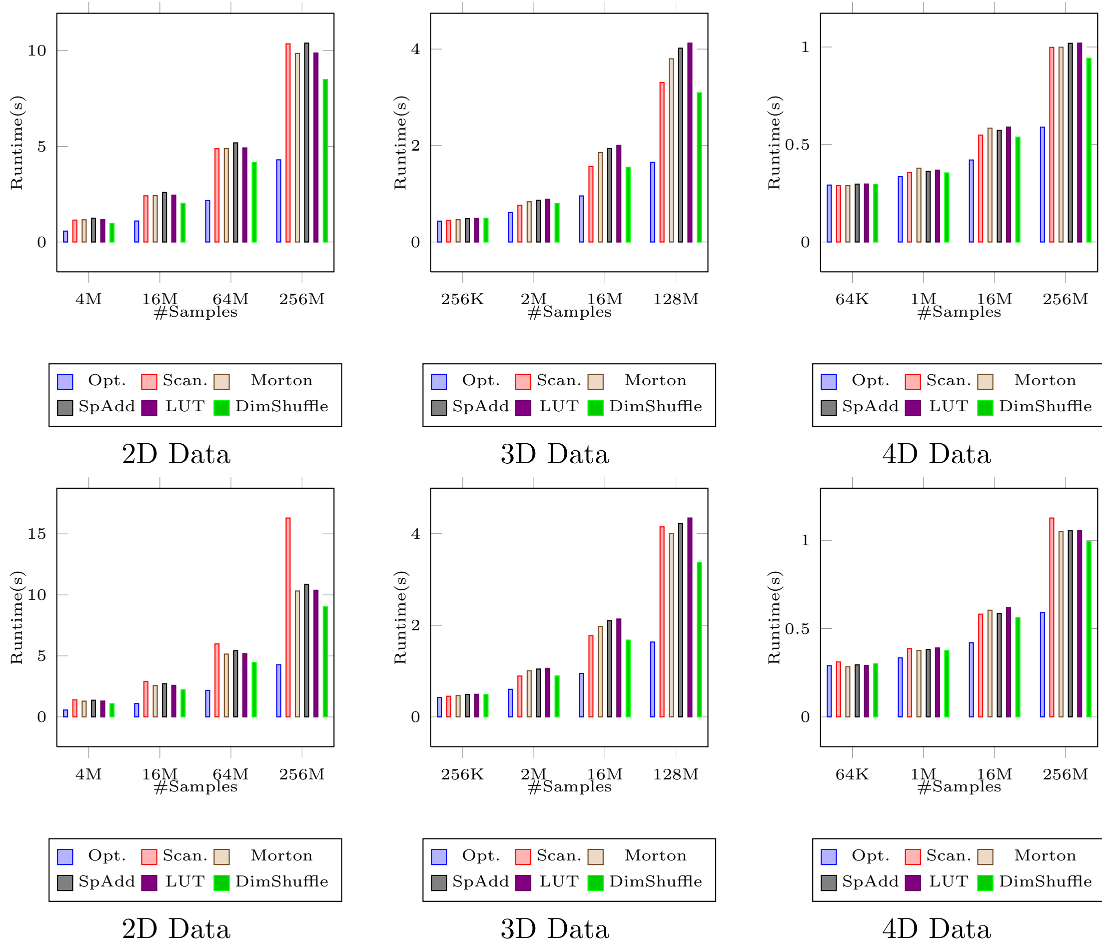


Figure 3.9: Total run-time in seconds for line integral computation. Top row: single-threaded; Bottom row: quad-threaded.

watts. This is due to the similar power consumption by L1 and L2 caches and the processor core. The L3 cache is expected to spend more energy but it is a very small portion of the program which has minimal effect on the measured power consumption. With similar power consumption, the total used energy to run the code is proportional to the run-time.

Unfortunately, the tested system does not report the energy spent by main memory data access. Since the data access requests to main memory is similar to L3 cache in the simulations, we expect large improvements in memory energy consumption.

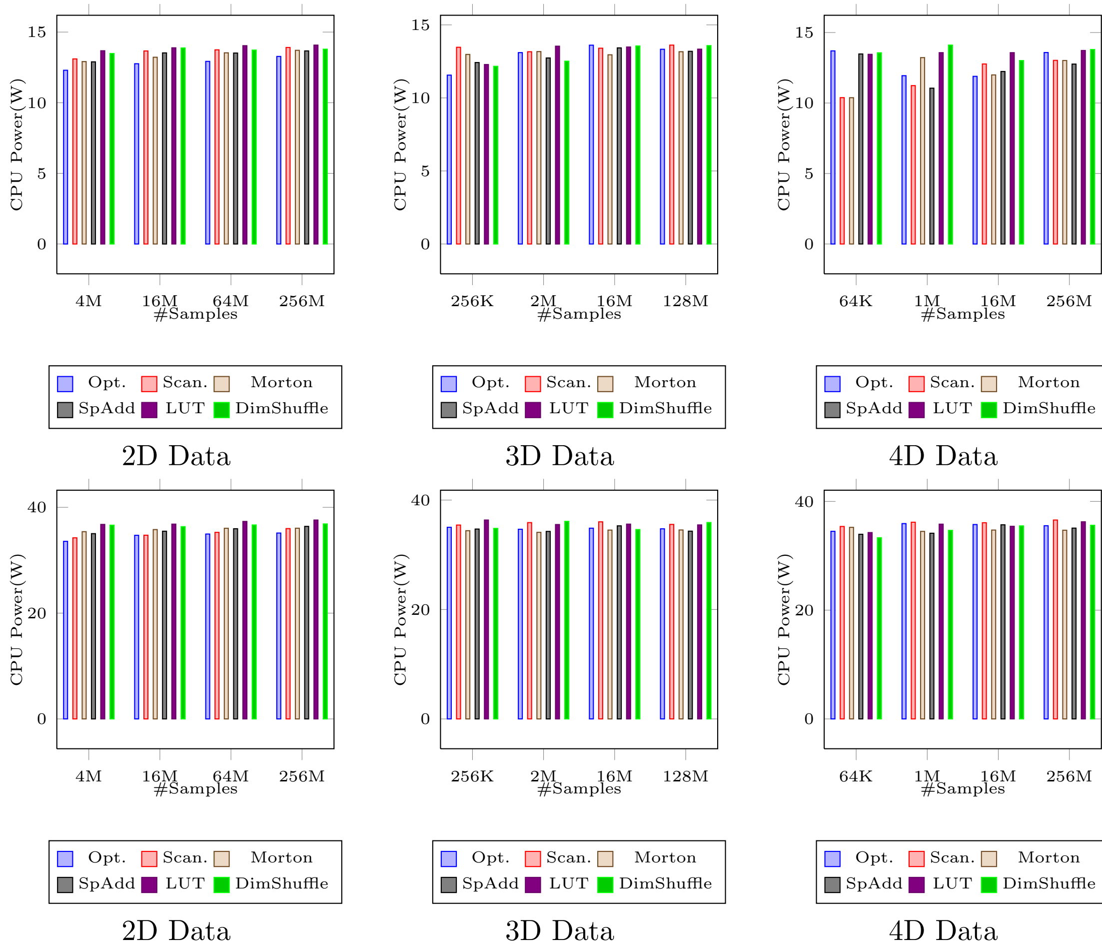


Figure 3.10: Total Power (W) used by the processor package for line integral computation. Top row: single-threaded; Bottom row: quad-threaded.

3.3 Chapter Summary

The performance of data reordering depends on two factors: cache utilization and index calculation. In many cases there is a trade-off between these two factors. A data ordering with better spatial locality usually involves more complex index calculation. With relaxing the spatial locality criteria as explained, we can achieve reasonable index calculation overhead without sacrificing much cache performance. With this idea, we defined `DimShuffle` indexing that outperforms `Scanline` in runtime and energy consumption with a more reliable margin.

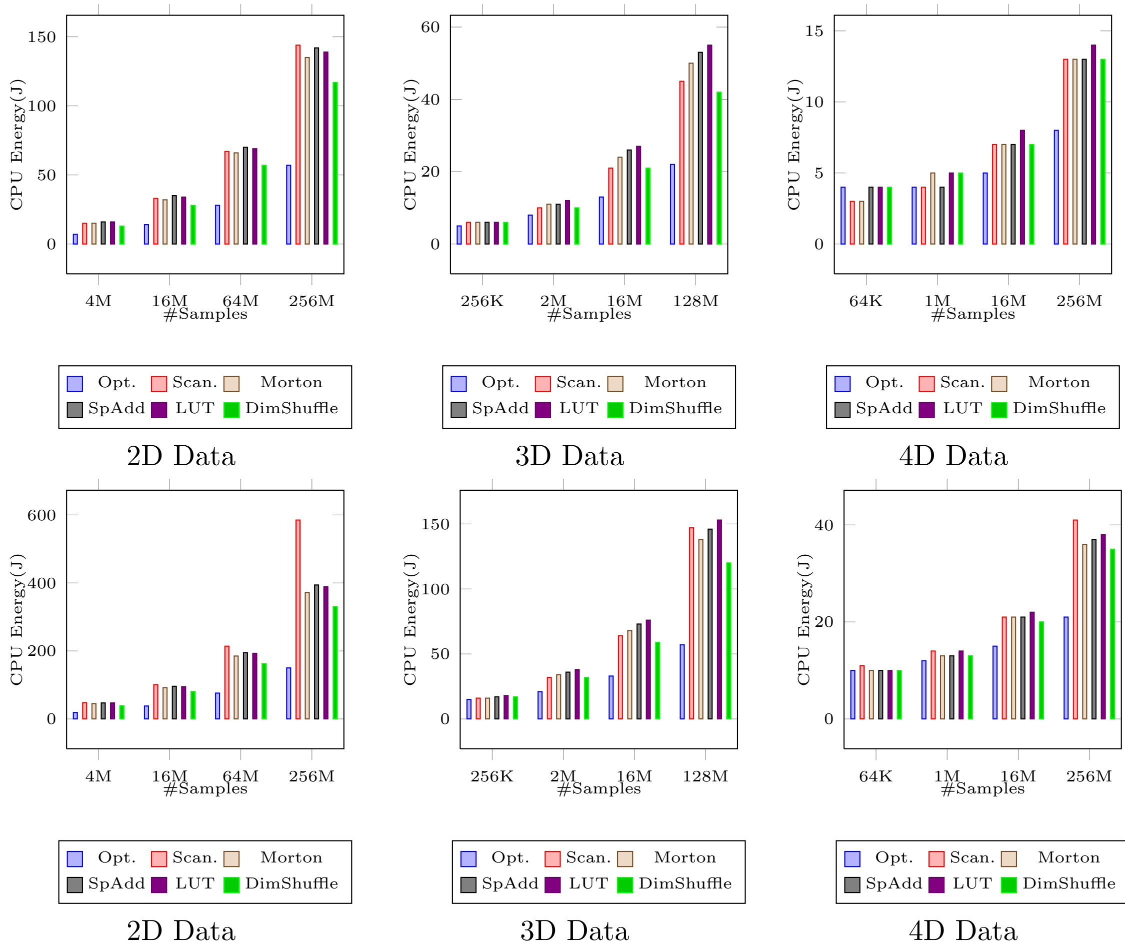


Figure 3.11: Total energy (J) used by the processor package for line integral computation. Top row: single-threaded; Bottom row: quad-threaded.

We have also shown the effect of parallel execution on cache behavior and system performance. Although small data sizes and low memory traffic on shared cache and memory processor cores seem to have no noticeable effect on each other, this is not the case in extreme conditions. For large data sizes, where data does not fit in the cache, and spatial locality is highly distorted, the shared bandwidths can be saturated. The race between cores to fetch data from shared memory reduces parallelization speedup. In such case the efficiency of the prefetcher also becomes extremely important since it is using the shared bandwidth for data that might not be ever used. This condition increases the importance of high data locality in runtime and energy consumption optimization.

Chapter 4

Conclusion and Future Work

In this work we studied cache utilization optimization for n-dimensional arrays with data reordering. We used row-major ordering for 2/3/4 dimensional data as the baseline. This ordering is the default in many compilers such as C/C++ and Java. The tested algorithm is a line-integral calculation for random lines in an n-dimensional hyper-cube. Due to the similarity of the dimensions in the application, row-major and column-major ordering has similar performance and we call these orderings under the general name of `Scanline`. We evaluated the performance of Morton ordering with direct index calculation (`Morton`), Morton ordering with index update (`SpAdd`), hybrid Morton-Scanline with look-up table (`LUT`), and dimension-shuffle block ordering with index update (`DimShuffle`).

Our contribution to the data reordering problem was to relax the global spatial locality constrain to a small set of block sizes. Our simulations confirm that data locality should be optimized for blocks with the size of different cache lines and pages. This gives the programmer the opportunity to design new data reordering schemes with easier index calculation. Our `LUT` and `DimShuffle` method use the relaxed definition of data locality, with the same cache performance of Morton ordering, which targets global data locality. Our `DimShuffle` ordering outperforms all other methods due to the ease of index calculation and high cache utilization.

Although Morton ordering has long been known as a way to improve cache utilization, there has never been solid proof for its overall run-time performance. As a result, many people just ignore the optimization and simply use the straightforward Scanline ordering. Our work opens the possibilities to design more computationally efficient indexing methods not only for Cartesian lattices, but also for general lattice types. We replaced the task

of minimizing the spatial locality distortion globally with a simpler target: maximize the spatial locality of the data stored in one page or in one cache line. The ordering of data within a block is not important, nor is the ordering of the blocks.

The performance improvement with data reordering depends on many factors, including memory to processor latency ratio, cache size and associativity, bandwidth and so on. We expect higher improvement on multi-core mobile processors such as the Intel Atom and the ARM Cortex-A series. We also expect a noticeable improvement in the energy spent by the main memory, which we could not measure in our experiments.

As future work, we see several possible extensions to this project. Our relaxed data reordering criteria can be used to define efficient data structures for lattices in specific applications. Defining a properly sized block and achieving data locality within this block is all that is needed for an optimal performance. Studying the effect of system parameters can help compilers find efficient data reordering schemes automatically, similar to what the OPIE compiler [17] does for Morton ordering.

Appendix A

Hardware Performance Counters

This chapter summarizes the hardware performance counters and event types we used in our experiments. These events are specific to the Intel SandyBridge cores i5-2300 processors. Since Intel is not always consistent with the names it might not be directly useful for other processors.

A.1 Performance Events

Var.	Event Type	Description
FIX0	CPU_CLK_UNHALTED_CORE	The frequency cycles the core is active
FIX1	CPU_CLK_UNHALTED_REF	Counts the frequency cycle; basically a timer
FIX2	INSTR_RETIRED_ANY	Retired (Finished) instructions
PMC0	DTLB_LOAD_MISSES_CAUSES_A_WALK	L1 Data TLB misses
PMC1	MEM_UOP_RETIRED_LOADS	Retired load instructions
PMC2	MEMLOAD_UOPS_RETIRED_L1_HIT	Load micro-ops. with L1 cache hit
PMC3	MEMLOAD_UOPS_RETIRED_HIT_LFB	Load micro-ops. with L1 cache miss, where the data is being requested before by earlier miss
PMC4	MEMLOAD_UOPS_RETIRED_L2_HIT	Load micro-ops. with L2 cache hit

PMC5	MEMLOAD_UOPS_RETIRED_ LLC_HIT	Load micro-ops. with LLC (L3) cache hit
PMC6	MEM_LOAD_UOPS_MISC_RETIRED_ LLC_MISS	Load micro-ops. with LLC (L3) cache miss with unknown information as data source
PMC7	L1D_REPLACEMENT	Number of lines brought into L1 cache
PMC8	L2_TRANS_DEMAND_ DATA_RD	Demand data read requests that access L2 cache
PMC9	L2_RQSTS_ALL_DEM_AND_DATA_RD	Demand and L1 HW. prefetch data read requests that access L2 cache
PMC10	OFFCORE_REQUESTS_DEMAND_ DATA_RD	Demand data read requests that access OffCore (L3) cache
PMC11	L2_TRANS_ALL_REQUESTS	All transactions to L2 cache
PMC12	L2_RQSTS_MISS	All L2 cache misses
PMC13	L2_LINES_IN_ALL	Number of lines brought into L2 cache
PWR0	PWR_PKG_ENERGY	CPU package energy packets
PWR2	PWR_DRAM_ENERGY	DRAM energy packets

A.2 Performance Metrics

Table A.2: The list of performance metrics we used, with their calculation formula. Metrics in the same block are measured in one run.

Performance Metric	unit	Equation
Runtime (RDTSC)	sec	time
Runtime unhalted	sec	FIXC1*inverseClock
Clock	MHz	1.0E-09*(FIXC0/FIXC1)/inverseClock
CPI		FIXC0/FIXC2
Data Rd. L1 Hit	GB	1.0E-09*PMC2*16.0
Data Rd. LFB Hit	GB	1.0E-09*PMC3*16.0
Data Rd. L2 Hit	GB	1.0E-09*PMC4*16.0
Data Rd. L3 Hit	GB	1.0E-09*PMC5*16.0
Data Rd. L2-to-L1 BW.	GBytes/s	1.0E-09*PMC7*64.0/time
Data Rd. L2-to-L1 Vol.	GBytes	1.0E-09*(PMC7)*64.0
Data Rd. L2 BW.	GBytes/s	1.0E-09*PMC8*64.0/time
Data Rd. L2 Vol.	GBytes	1.0E-09*PMC8*64.0
Data+Code Rd. L3-to-L2 BW.	GBytes/s	1.0E-09*PMC12*64.0/time
Data+Code Rd. L3-to-L2 Vol.	GBytes	1.0E-09*PMC12*64.0
L2 request rate		PMC11/FIXC2
L2 miss rate		PMC12/FIXC2
L2 miss ratio		PMC12/PMC11
L2 hit ratio		1-(PMC3/PMC11)
Data Rd.+Pref. L2 BW.	GBytes/s	1.0E-09*PMC9*64.0/time
Data Rd.+Pref. L2 Vol.	GBytes	1.0E-09*PMC9*64.0
Data+Code Rd.+Pref. L3 BW.	GBytes/s	1.0E-09*PMC13*64.0/time
Data+Code Rd.+Pref. L3 Vol.	GBytes	1.0E-09*(PMC13)*64.0
Data Rd. OffCore (L3) BW.	GBytes/s	1.0E-09*PMC10*64.0/time
Data Rd. OffCore (L3) Vol.	GBytes	1.0E-09*PMC10*64.0
Data. Misc. Rd. L3 Miss BW.	GBytes/s	1.0E-09*PMC6*64.0/time
Data. Misc. Rd. L3 Miss Vol.	GBytes	1.0E-09*PMC6*64.0
Data Rd. BW.	GBytes/s	1.0E-09*PMC1*16.0/time
Data Rd. Vol.	GBytes	1.0E-09*(PMC1)*16.0
Data Rd. L2 Requests	GBytes	1.0E-09*(PMC1-PMC2-PMC3)*16.0
L1 DTLB miss rate		PMC0/FIXC2
CPU Energy	J	PWR0
CPU Power PKG	W	PWR0/time

Bibliography

[1] *2nd Generation Intel Core Processor Family Desktop Datasheet*, volume 1. Intel Corporation, November 2012.

[2] *2nd Generation Intel Core Processor Family Desktop Datasheet*, volume 2. Intel Corporation, November 2012.

[3] *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, November 2012.

[4] *Intel Core Processors Technical Resources.* Intel Corporation, November 2012.

[5] Space-filling curve. http://en.wikipedia.org/w/index.php?title=Space-filling_curve&oldid=522177923, November 2012. Page Version ID: 522177923.

[6] Michael D. Adams and David S. Wise. Fast additions on masked integers. *SIGPLAN Not.*, 41(5):39–45, May 2006.

[7] Roch G. Archambault, Robert J. Blainey, and Yaoqing Gao. Compiler with cache utilization optimizations, July 2010. U.S. Classification: 717/154.

[8] Evangelia Athanasaki and Nectarios Koziris. Improving cache locality with blocked array layouts. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 308–317, Athens, Greece, February 2004.

[9] Evangelia Athanasaki and Nectarios Koziris. Fast indexing for blocked array layouts to reduce cache misses. *International Journal of High Performance Computing and Networking*, 3(5):417–433, January 2005.

[10] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 486–500, Sorrento, Italy, 2001. ACM.

[11] John M. Calandrino and James H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Euromicro Conference on Real-Time Systems*, pages 299–308, July 2008.

[12] Hui Chen, Shinan Wang, and Weisong Shi. Where does the power go in a computer system: Experimental analysis and implications. In *International Green Computing Conference and Workshops (IGCC)*, pages 1–6, July 2011.

[13] Trishul M. Chilimbi, James R. Larus, and Robert Davidson. Data structure partitioning to optimize cache utilization, December 2001. U.S. Classification: 1/1 International Classification: :G06F 1730.

[14] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):599–608, August 2010.

[15] Hossam ElGindy and George Ferizis. On improving the memory access patterns during the execution of Strassen's matrix multiplication algorithm. In *Proceedings of the 27th Australasian Conference on Computer Science*, volume 26 of *ACSC '04*, pages 109–115, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[16] Michael Franz and Thomas Kistler. Splitting data objects to increase cache utilization. Technical report, Department of Information and Computer Science, University of California Irvine, CA, USA, October 1998.

[17] Steven T. Gabriel and David S. Wise. The OPIE compiler from row-major source to Morton-ordered matrices. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 136–144, Munich, Germany, 2004. ACM.

[18] Allan Gottlieb. Class notes for computer architecture. http://cs.nyu.edu/{˜}gottlieb/courses/2000-01-fall/arch/lectures/lecture-22.html, 2000.

[19] Sren Grimm, Stefan Bruckner, Armin Kanitsar, and Eduard Grller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers & Graphics*, 28(5):719–729, October 2004.

[20] Frank Gnther, Miriam Mehl, Markus Pgl, and Christoph Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on spacefilling curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, January 2006.

[21] Jos R. Herrero. New data structures for matrices and specialized inner kernels: Low overhead for high performance. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, number 4967 in Lecture Notes in Computer Science, pages 659–667. Springer Berlin Heidelberg, January 2008.

[22] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactons on Computer Systems*, 10(4):338–359, November 1992.

[23] Gunter Knittel. The ULTRAVIS system. In *IEEE Symposium on Volume Visualization*, pages 71–79, October 2000.

[24] James R. Larus, Robert Davidson, and Trishul M. Chilimbi. Field reordering to optimize cache utilization, March 2002. U.S. Classification: 717/159 International Classification: :G06F/945.

[25] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[26] Lixia Liu, Zhiyuan Li, and Ahmed H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 359–367, Aegean Sea, Greece, June 2008. ACM.

[27] K. Patrick Lorton and David S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Computer Architecture News*, 35(4):6–12, September 2007.

[28] Prasanth Mangalagiri, Karthik Sarpatwari, Aditya Yanamandra, VijayKrishnan Narayanan, Yuan Xie, Mary Jane Irwin, and Osama Awadel Karim. A low-power phase change memory based hybrid cache architecture. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 395–398, Orlando, FL, USA, May 2008. ACM.

[29] Sally A. Mckee and William A. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 253–262, Charlottesville, VA, USA, 1995.

[30] G. M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[31] Khoa Nguyen, Wei Hsu, and Hui-May Chang. Method of efficient dynamic data cache prefetch insertion. Publication number: US 2003/0145314 A1 U.S. Classification: 717/158 International Classification: :G06F009/45; G06F012/00.

[32] Anthony E. Nocentino and Philip J. Rhodes. Optimizing memory access on GPUs using Morton order indexing. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 18:1–18:4. ACM, 2010.

[33] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. Controlling cache utilization of HPC applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 295–304, New York, NY, USA, June 2011. ACM.

[34] Rajeev Raman and David S. Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, April 2008.

[35] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, August 2006.

[36] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, February 2007.

[37] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul Kelly. Improving the performance of morton layout by array alignment and loop unrolling. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 241–257. Springer Berlin / Heidelberg, 2004.

[38] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul Kelly. Is morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience*, 18(11):1509–1539, January 2006.

[39] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In *39th International Conference on Parallel Processing Workshops*, pages 207–216, September 2010.

[40] David Wise. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 774–783. Springer Berlin / Heidelberg, 2000.

[41] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. *Proceedings of the 8th ACM Symposium on Principles and Practices of Parallel Programming*, 36(7):24–33, June 2001.

[42] Honesty C. Young and Eugene J. Shekita. An intelligent I-cache prefetch mechanism. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 44–49, San Jose, CA, USA, October 1993.

[43] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(2):407–425, May 2004.

[44] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 136–146, June 2003.