Overcoming the Memory Wall in Symbolic Algebra: A Faster Permutation Multiplication

Gene Cooperman* and Xiaoqin Ma*
gene,xqma@ccs.neu.edu
Northeastern University
Boston, MA 02115
{gene,xqma}@ccs.neu.edu

January 9, 2003

Abstract

The traditional permutation multiplication algorithm is now limited by memory latency and not by CPU speed. A new cache-aware permutation algorithm speeds up permutation multiplication by a factor of 3.4 on current CPUs. The new algorithm is limited by memory bandwidth, but not by memory latency. Current trends indicate improving memory bandwidth and stagnant memory latency. This makes the new algorithm especially important for future computer architectures. In addition, we believe this "memory wall" will soon force a redesign of other common algorithms of symbolic algebra.

1 Introduction

The problem of a large CPU-memory gap was popularized by Wulf and McKee [3], who introduced the term *memory wall*. Our own work is motivated by the experience of the first author with the memory wall in Cooperman and Grinberg [2]. In that work, they designed a parallel shared memory algorithm that should have achieved a linear speedup with the number of processors. Yet the experimentally measured speedup never went beyond a factor of seven. This was conclusively attributed to limitations of memory latency.

In this article, we demonstrate that the classical two-line sequential algorithm for permutation multiplication has now hit this same memory wall. On a Pentium 4 with PC-133 RAM, permutation multiplication is independent of the CPU speed, and is limited by memory latency. We also demonstrate a new permutation algorithm that more than doubles the speed of the traditional algorithm by making a detour around the memory wall.

To illustrate the impact of the memory wall, consider a 1.7 GHz Pentium 4 with PC-133 RAM. A consecutive access to the next 4-byte integer in RAM costs 6.4 CPU cycles and a random access incurring a cache miss costs 255 CPU cycles. To see this, note that the 64-bit memory bus of a Pentium with PC-133 RAM implies 8×133 = 1.064 GB/s memory bandwidth. So, consecutive access requires $1.7 \times (4/1.064) \approx 6.4$ CPU cycles. For the 128 byte cache of the Pentium 4, the 255 CPU cycles derives from 128/1.064 ns to load the cache block, plus 30 ns SDRAM latency for accessing the PC-133 in a non-sequential manner. These numbers have been verified both experimentally and from the PC-133 RAM specifications [1, Sections 4.1,4.2,4.5,4.6].

As the CPU-memory gap continues to widen, we predict that other symbolic algebra programs will also experience the memory wall. This is because symbolic algebra programs, unlike numerical analysis programs, often access large amounts of main RAM in a pseudo-random fashion. Thus symbolic algebra programs present a worst case scenario for current RAM technology.

*This work was partially supported by the National Science Foundation under Grant CCR-0204113, and by the Institute for Complex Scientific Software (ICSS, http://www.icss.neu.edu/).

2 Definitions

Recall that a *permutation* on n points is a one-to-one and onto mapping ϕ from the integers $\{1, 2, \ldots, n\}$ to itself. In C/C++, a permutation on n points is most naturally represented as an array of integers, `X[]`, for which the range contains all values from 1 to n. If $\phi(i) = j$, then we represent this as `X[i] == j`.

Given two permutation mappings, $i \mapsto \phi(i)$ and $i \mapsto \gamma(i)$, a *permutation multiplication* is the composition of the two mappings: $i \mapsto \gamma(\phi(i))$. The C/C++ version of this is:

```
int X[ARRAY_SIZE], Y[ARRAY_SIZE], Z[ARRAY_SIZE];
for (i = 0; i < ARRAY_SIZE; i++) Z[i] = Y[X[i]];
```

3 The New Permutation Multiplication Algorithm

The key to a faster permutation multiplication algorithm is to replace arbitrary accesses to RAM by sequential accesses to RAM. The code for the new *Localized Algorithm* follows in Figure 1. The `ARRAY_SIZE` (as the number of integers) must be specified along with the L2 `CACHE_SIZE` of the target computer.

```
#define BLOCK_LENGTH (CACHE_SIZE/2/sizeof(int))
#define NUMBER_OF_BLOCKS (ARRAY_LENGTH / BLOCK_LENGTH)
int X[ARRAY_LENGTH], Y[ARRAY_LENGTH], Z[ARRAY_LENGTH];
int D[ARRAY_LENGTH];
int * D_ptr[NUMBER_OF_BLOCKS];

//Phase I: distribute value, X[a], into D_ptr[block_num]
//           such that block_num == X[a] / BLOCK_LENGTH
int block_num, i, j;
for (block_num= 0; block_num < NUMBER_OF_BLOCKS; block_num++)
  D_ptr[block_num] = & D[block_num * BLOCK_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++) {
  block_num = X[i] / BLOCK_LENGTH;
  * D_ptr[block_num] = X[i];
  D_ptr[block_num]++;
}

//Phase II: for D[i] == X[a], replace the value X[a] by Y[X[a]]
//  Note that |i - D[i]| == |i - X[a]| and |i-X[a]| < BLOCK_LENGTH
for (i = 0; i < ARRAY_LENGTH; i++)
  D[i] = Y[ D[i] ];

//Phase III: copy value Y[X[a]] from D_ptr[block_num] to Z[a]
for (block_num = 0; block_num < NUMBER_OF_BLOCKS; block_num++)
  D_ptr[block_num] = & D[block_num * BLOCK_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++) {
  block_num = X[i] / BLOCK_LENGTH;
  Z[i] = * D_ptr[block_num];
  D_ptr[block_num]++;
}
```

Figure 1: Localized Algorithm

The algorithm requires several preconditions. These are `block_size` $\leq$ `CACHE_SIZE/2`, `number_of_blocks` $\leq$ `number_of_cache_lines`, and the arrays should fit in RAM. Where the preconditions are not met, a two-level version of this same algorithm can be designed. The use of the array of pointers, `D_ptr`, instead of the more natural multi-dimensional arrays was dictated by reasons of efficiency.

How Does it Work? The inputs are the `X` array and `Y` array. The output is the `Z` array. Conceptually, one should imagine the `D` array and the `Y` array as each being divided into blocks or bins, where `D[i]` is in bin `block_num` if $\texttt{block\_num} \leq i/\texttt{BLOCK\_SIZE} < \texttt{block\_num} + 1$.

Phase I. This phase proceeds as in binsort. Each element, `D_ptr[block_num]`, is initialized to point to the beginning of a contiguous block or bin of the `D` array. Each bin holds a range of contiguous values. The values in the `X` array are then distributed into the bins. All accesses to RAM occur as accesses to `X`, or to `D_ptr[block_num]` for each `block_num`. Accesses in each such data stream are sequential.

For example, assume an X array = {0,7,10,2,4,9,3,6,8,1,5,11}, a `Y` array = {1000,1001,1002,1003,1004, 1005,1006,1007,1008,1009,1010,1011}, and assume that a block of the `D` array hold 4 integers. Then at the end of this phase we have D[0] = {0,2,3,1}, D[1] = {7,4,6,5}, D[2] = {10,9,8,11} .

Phase II. In this phase, each value `X[a]` in the `D` array is replaced by a value `Y[X[a]]`. Because `|i - X[a]| < BLOCK_SIZE`, the value `Y[X[a]]` is close to `Y[i]` and so `Y[X[a]]` is likely to be present in cache. Further, the value of `BLOCK_SIZE` was chosen small enough so that corresponding bins from both the `D` array and the `Y` array could be simultaneously resident in cache.

After Phase II, the `D` array is D[0] = {Y[0],Y[2],Y[3],Y[1]} = {1000,1002,1003,1001}, D[1] = {Y[7],Y[4], Y[6],Y[5]} = {1007,1004,1006,1005}, D[2] = {Y[10],Y[9],Y[8],Y[11]} = {1010,1009,1008,1011}.

Phase III. The `D` array contains data of the form `Y[X[a]]`. It is copied into the target `Z` array. In Phase I, the elements of X were copied into a given bin of `D` in the order in which they occurred in `X`. We use that fact to copy the data `Y[X[a]]` of `D` into the `Z` array in that same order. If the element `X[a]` of the `X` array was placed into the k^{th} bin of the `D` array at step `a` of Phase 1, then the element $Y[X[a]]$ will be copied from the k^{th} bin of the `D` array into `Z[a]` at step `a` of Phase III. As in Phase I, accesses within each of the data streams are sequential.

Since `X[1] = 7` was placed into the 2^{nd} block of the `D` array at step 1 of Phase I, then `Z[1]` will come from the 2^{nd} block of the `D` array at step 1 of Phase III. As a result, the `Z` array will be
Z = {Y[0],Y[7],Y[10],Y[2],Y[4],Y[9],Y[3],Y[6],Y[8],Y[1],Y[5],Y[11]}
= {1000,1007,1010,1002,1004,1009,1003,1006,1008,1001,1005,1011}.

Although we simultaneously read and write from multiple streams (since each bin `D[block_num]` acts as a separate stream), the Pentium 4 and PC-133 hardware provide the performance benefits as if each stream were accessed sequentially without interference from other streams. For architectures where this is not the case, this can be enforced by the use of assembly instructions for prefetching and cache line flush. After reading/writing an element in a stream, one prefetches the next several elements of that stream to gain the benefits of sequential access. Such instructions are available in most microprocessor architectures.

4 Experimental Results

Table 1 shows overall times for the product of two random permutations on 1,048,576 points on some lightly loaded computers for different architectures. All experimental times were repeatable to within 2%. Although the different computers had available different versions of the C compiler, we also compiled the source into assembly code under gcc-2.95 and copied that assembly code to all computers. All times were again within 2% of the original times.

The Pentium II has an off-chip 512 KB L2 cache and so `CACHE_SIZE` is set to 512K/sizeof(int) (128K int's). The Pentium III and Pentium 4 have an on-chip 256 KB L2 cache and so `CACHE_SIZE` is set to 256K/sizeof(int) (64K int's) for the new algorithm. The Linux 2.4 kernel was used in all cases.

CPU/RAM	Compiler	Time (new algo.)	Time (traditional algo.)
2.66 GHz Pentium 4 / DDR-266 RAM	gcc-3.3 -O2	0.047 s	0.159 s
1.7 GHz Pentium 4 / PC-133 RAM	gcc-3.2 -O2	0.076 s	0.171 s
0.6 GHz Pentium III / PC-100 RAM	gcc-2.96 -O2	0.149 s	0.097 s
0.35 GHz Pentium II / PC-100 RAM	gcc-2.95 -O2	0.258 s	0.148 s

Table 1: Multiplication of two random permutations on 1,048,576 points (4 MB per permutation)

It may seem surprising that the Pentium 4 is slower on the traditional algorithm than is the Pentium III. This can be understood by noting that Intel raised the cache line size on the Pentium 4 from 32 bytes to 128 bytes, in order to improve the memory bandwidth. Hence, on a cache miss, the Pentium 4 pays a penalty four times as large.

Table 1 also provides strong evidence that the traditional permutation algorithm is now limited by the speed of RAM and not by the speed of the CPU. Recall from Section 1 that for the Pentium 4 with PC-133 RAM, the traditional permutation multiplication incurs a 150 ns random access time for each element of the Y array. This yields a lower estimate of 0.15 s for the overall CPU time of the traditional algorithm. This number is close to the actual CPU time of 0.171 s reported in Table 1, indicating that most of the CPU time is consumed by waiting on memory.

5 Generalization to other permutation operations

Permutation multiplication is but one of three common permutation operations that must be efficiently implemented in programs relying on permutations. The other two are *permutation inverse*

```
for (i = 0; i < ARRAY_SIZE; i++) Z[X[i]] = i;
```

and *multiplication by an inverse*

```
for (i = 0; i < ARRAY_SIZE; i++) Z[X[i]] = Y[i];
```

The ideas of this paper generalize to the two other operations. For example, in order to efficiently compute a permutation inverse, one need only copy the pair `(X[i], Y[j])` to the appropriate block in the D array.

6 Conclusion

In this paper, we have examined and re-designed one of the simplest computer algorithms, *permutation multiplication*. It gains greater data locality in exchange for having to make two passes. On a 2.66 GHz Pentium 4 with PC-133 RAM, the new algorithm is 3.4 times as fast as the traditional one. As the CPU-memory gap grows, the new algorithm is expected to show a still greater improvement over a traditional permutation multiplication.

As one might expect, the new algorithm can also be employed elsewhere in the memory hierarchy. For example, the new algorithm provides the first practical method for multiplying large permutation arrays on disk.

The algorithm also has important extensions to non-permutations. When the Y array is an array of data objects, Y[X[i]] can be viewed as a way of reorganizing the order of the data objects according to the ordering X[i]. Such reordering of data objects is employed throughout computer science. These ideas and others will be developed in a later paper.

References

[1] *Intel PC SDRAM Specification, Revision 1.7, November 1999.* `http://www.intel.com/technology/memory/pc133sdram/spec/sdram133.htm`

[2] G. Cooperman and V. Grinberg, “Scalable Parallel Coset Enumeration: Bulk Definition and the Memory Wall”, *J. Symbolic Computation* **33**, pp. 563–585.

[3] W.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, 1995.