Electronic Transactions on Numerical Analysis.
Volume 15, pp. 66-77, 2003.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

CACHE AWARE DATA LAYING FOR THE GAUSS-SEIDEL SMOOTHER*

MALIK SILVA†

Abstract. Feeding the processor with data operands is the bottleneck in many scientific computations. This bottleneck is alleviated by means of caches, small fast memories to keep data. The performance of a memory-intensive computation depends critically on whether most of the data accesses can be performed within the cache. Thus, cache aware computing is of importance. There are several well established strategies available to a programmer to make a program cache friendly. In this paper, we describe cache aware data laying, a technique which we feel has not been researched sufficiently. It is a promising technique as we achieved considerable performance improvements. For example, our data laying experiments with the Gauss-Seidel smoother resulted in up to 84% execution time improvements over the parallelogram based blocked implementation of the algorithm.

Key words. memory barrier, caches, iterative algorithms, cache-aware algorithms, data laying

AMS subject classifications. 65, 68I

1. Introduction.

1.1. Memory Barrier. The gap between processor and memory speeds continues to widen as processor speeds are increasing at a rate far greater than memory speeds. The increases in processor speeds have not been accompanied by similar increases in memory speeds and unfortunately, this performance gap is projected to continue increasing. Although the memory storage capacity has improved tremendously, the access times have improved only slowly. The latency of a memory system is typically a factor of 50 slower than the clock rate of the processor. Furthermore, there is also a bandwidth problem with memory. For example, the total bandwidth required to sustain the peak performance of an Alpha 21164 processor is around 24 Gbytes/sec, but in contrast to this, typical memory systems built around an Alpha processor have a nominal peak bandwidth of less than 1 Gbyte/sec [8]. Thus, there is a barrier to performance at memory. This barrier is a major hindrance to computer performance. It has become a dominant bottleneck in overall application execution time and acts as an obstacle to achieving the full benefits that are attainable using current processors. To fully realize the potential of the processors, it is very important to overcome this memory barrier. This is furthermore important as the predictions for technological trends for the coming 15 years [8] indicate this performance gap widening.

1.2. Caches. Present day computers have a hierarchy of memory as a way of reducing this memory barrier and providing a reasonable amount of fast memory that the programmers desire. The hierarchy of memory consists of small fast memory at the top of the hierarchy which is closest to the processor action but getting larger and slower down the hierarchy, further away from the processor action. The present generation of processors usually include two levels of caches: a fast, small, on-chip, level 1 (L1) cache and a relatively slower but larger, off-chip, level 2 (L2) cache. L1 cache, though small, is very fast. L2 cache is relatively slower but is still faster than memory access. The smallest unit of data that can be transferred between the memory and the cache system is called a *cache-line*. The goal is to place the cache-lines that the processor needs in the fast L1 cache, or at least in the L2 cache. Then, only the addresses that miss in the L2 cache have to be fetched from memory. Thus the cache hierarchy can be a good solution to the memory barrier. Given that the cache is faster than

*This work was supported by the Swedish International Development Agency - IT Project. Received May 23, 2001. Accepted for publication September 12, 2001. Recommended by Craig Douglas.
†Department of Computer Science, University of Colombo and Department of Scientific Computing, Uppsala University (`malsil@tdb.uu.se`).

ETNA
Kent State University
etna@mcs.kent.edu

main memory in fulfilling a processor's memory requests, one clearly needs to maximize the number of cache hits which are the accesses that can be satisfied from the cache.

1.3. Cache Misses. Unfortunately, due to the limited size of the cache, three types of cache misses occur in a single processor system: *compulsory*, *capacity*, and *conflict*[3]. Compulsory misses are the misses that occur for each first reference of a cache-line containing a data item. Capacity miss is a miss that is not a compulsory miss but which misses in a fully associative cache. That is due to the limited size of the cache. Conflict miss is a reference that hits in a fully associative cache but misses in a set associative cache. That is due to addresses mapping to identical cache blocks. Cache conscious programmers have to try to avoid all cache misses whenever possible.

1.4. Necessity for Cache Aware Programs. When the data set of a computation is large such that it no longer fits in the cache, cache aware computer programs become absolutely essential to achieving anything other than poor performance when measured against possible peak performance figures of the computers. For example, let us consider the red-black Gauss-Seidel as a smoother in the multigrid method. Multigrid methods [2] are among the most attractive algorithms for the solution of large sparse systems of equations that arise in the solution of elliptic partial differential equations. However, even simple multigrid codes on structured grids with constant coefficients cannot exploit a reasonable fraction of the floating point peak performance of current microprocessors. Furthermore, the performance typically drops dramatically with growing problem size. Reason for this poor performance has been found to be the poor cache performance of the smoother component, typically red-black Gauss-Seidel, of the multigrid which is the most time consuming of the whole multigrid method [7].

Let us examine the standard red-black Gauss-Seidel. This algorithm repeatedly performs one complete sweep through the grid from bottom to top updating all the red nodes and then performs another complete sweep updating all the black nodes. Assuming the grid is too large to fit in the cache, the data of the lower part of the grid is no longer in the cache after the red update sweep because it has been replaced by the grid points belonging to the upper part of the grid. Hence, the data must be reloaded from the slower main memory into the cache again. In this process newly accessed nodes replace the upper part of the grid points in the cache, and as a consequence they have to be loaded from the main memory once more at the time of their next update. The performance degrades further when the grid size increases. This performance bottleneck at memory is also seen in most other scientific applications.

In general, the cost of floating point operations is rapidly decreasing. Moving data is what makes computing expensive. Although there are different ways of organizing on-chip and off-chip memory [8], programmers will have to learn that reasonable performance of a system could only be expected with programs being aware of the reduced performance of the off-chip memory access. For the algorithmic development, data locality may therefore become a key issue.

1.5. This Research. Our long term research is to address the issues involved in reordering data accesses within an algorithm and in developing algorithmic variants for scientific computing that enable high cache performance. In the particular research work that we describe in this report, we study red-black Gauss-Seidel, which is an integral component of multigrid methods. We attempt to improve the performance of this application by making it cache aware using existing techniques. We also study the benefits achievable through cache aware data laying.

The rest of this paper is organized as follows. Section 2 gives the background to our research. Section 3 describes the technique of data laying. Section 4 gives our implementation

ETNA
Kent State University
etna@mcs.kent.edu

details and section 5 the results. Section 6 summarizes our conclusions and section 7 gives directions for further research.

2. Background.

2.1. Current Work.

To reduce the memory barrier, the cache residence of memory accesses should be improved. There are several techniques [3] that are used to achieve this goal. These techniques either involve algorithm or data structure changes. We outline them below.

Fusion. It is a method to improve the temporal locality of data. When the same data is used in a task during separate sections of code, it is better if possible, to bring them close to the same sections in the code. Then, the data that is being fetched into the cache can be used repeatedly before being swapped out.

Blocking. Blocking is also a technique which tries to improve temporal locality. Instead of operating on entire rows or columns of a matrix which may be too big to fit in the cache, blocked algorithms operate on sub-matrices or, *data blocks*. The goal is to maximize accesses to the data loaded into the cache before the data is replaced. The best blocking size depends both on the data and cache sizes.

Prefetching. Prefetching is fetching data into cache before the processor actually needs it. It can be accomplished with compiler flags, via programmer intervention, or by hardware [5]. Software-based prefetching requires a special processor instruction which can be used by the compiler or programmer to issue the load from main memory. Having cache-lines longer than one word is an example of hardware-based prefetching: data brought into cache is accompanied by surrounding data; if the data exhibits spatial locality, then the surrounding data has usefully been prefetched. One of the main difficulties with prefetching is its lack of portability due to high machine dependence. Although long cache-lines support successful prefetching when the data exhibits spatial locality, long cache-lines also tend to increase conflict misses due to associativity problems. Also, support from the programming languages for application programmers to effectively use the hardware prefetch instruction is limited or non-existent [11]. Thus prefetching, though beneficial, has its share of concerns.

Padding. Padding is a way to reduce the likelihood of conflict misses. Here, the data are padded by a small multiple of the cache-line length. This technique is widely used. Rivera et.al.,[4] describe a successful implementation of compiler assisted padding. It shows the value of padding as an inexpensive method of conflict miss elimination. Rivera et.al.,[6] also describe successful implementations of blocking and padding based 3d scientific computation optimizations. The main negative aspect of padding, like prefetching, is its poor portability: the correct padding size is highly machine dependent.

Loop Interchange. This is a technique in which the nesting of loops are interchanged so that access of data from memory is in the order that they are stored. The misses are reduced by improving spatial locality.

Array Merging. This technique reduces misses by improving spatial locality. Some programs reference multiple arrays in the same dimension with the same indices at the same time. The danger is that these accesses will interfere with one another, leading to conflict misses. This danger is removed by combining these independent matrices into a single compound array so that a single cache block can contain the desired elements. As an example of the use of this technique, consider the following loop:

```
For I=1,n do
      C(I) = A(I)+B(I)
```

```
        Endfor
```

In the extreme case when A(I),B(I) and C(I) each map to the same cache block, this block will have to be constantly emptied and refilled leading to cache thrashing. A fix for this problem is to merge A, B and C into a single array, R. Then,

```
        For I=1,n do
                R(3,I) = R(1,I)+R(2,I)
        Endfor
```

When R is stored in column-major order, this technique drastically reduces the number of cache misses. This technique is also highly portable.

2.2. Related Work. Considerable work has been done on cache aware computing. However, the technique of algorithm related data laying, we feel, has not been sufficiently studied. Most of the existing data laying schemes we found were based on space filling curves [12] and not on algorithms (see [9] for an example). This research was motivated by that observation.

3. Cache Aware Data Laying. Caches are motivated by two principles of locality: *temporal* and *spatial*. Temporal locality states that data required now by the processor will also be necessary again in the near future. Spatial locality states that if specific data is required, its neighboring data will also be referenced soon. The aim of any cache conscious programmer should be to ensure the occurrence of these types of locality so that the code suits the concepts on which the caches are based.

To ensure temporal and spatial locality, the programmer can use the techniques described in the previous section such as fusion, blocking, loop interchange, and array merging. To further improve performance, the programmer can lay the data in memory according to the access pattern. This is the technique of *data laying*.

The smallest block of data moved in and out of a cache is a cache-line. A cache-line holds data that is stored contiguously in memory. Thus, it is advantageous to lay contiguously in memory, the data that we are going to use in succession. This will ensure cache-lines which are packed with important data. Thus, the spatial locality of data is improved. Then, when a cache-line is brought into the cache it also effectively prefetches the data that is to be needed in the near future. If elements in a data block are laid by access, spatial locality is improved in addition to the normal increase in temporal locality that the blocking technique supports. Since spatial locality is improved, increases in cache-line size will also provide increased successful prefetches. Since many hardware systems also prefetch the neighbouring cache-line(s) when a particular cache-line is fetched, more important data will also be prefetched. Since these layouts will result in stride-1 accesses, they also work well with memory banks which form the memory systems of many computers. Thus, laying data contiguously in memory is a good approach for improved performance benefits. This technique is illustrated in detail in the next section.

4. Implementation. The data laying technique has the potential to be tried out on any application. This report describes our data layout optimization effort in respect of the 2d red-black Gauss-Seidel smoother.

We consider the two-dimensional red-black Gauss-Seidel relaxation method based on a 5-point discretization of the Laplace operator. The red-black Gauss-Seidel relaxation algorithm is based on the red-black ordering of the unknowns as shown in Fig. 4.1. A standard implementation of the red-black Gauss-Seidel algorithm iteratively passes once through the whole grid (for example from bottom to top) and updates all of the red points, then passes a second time through the whole grid and updates all of the black points. The layout of grid

ETNA
Kent State University
etna@mcs.kent.edu

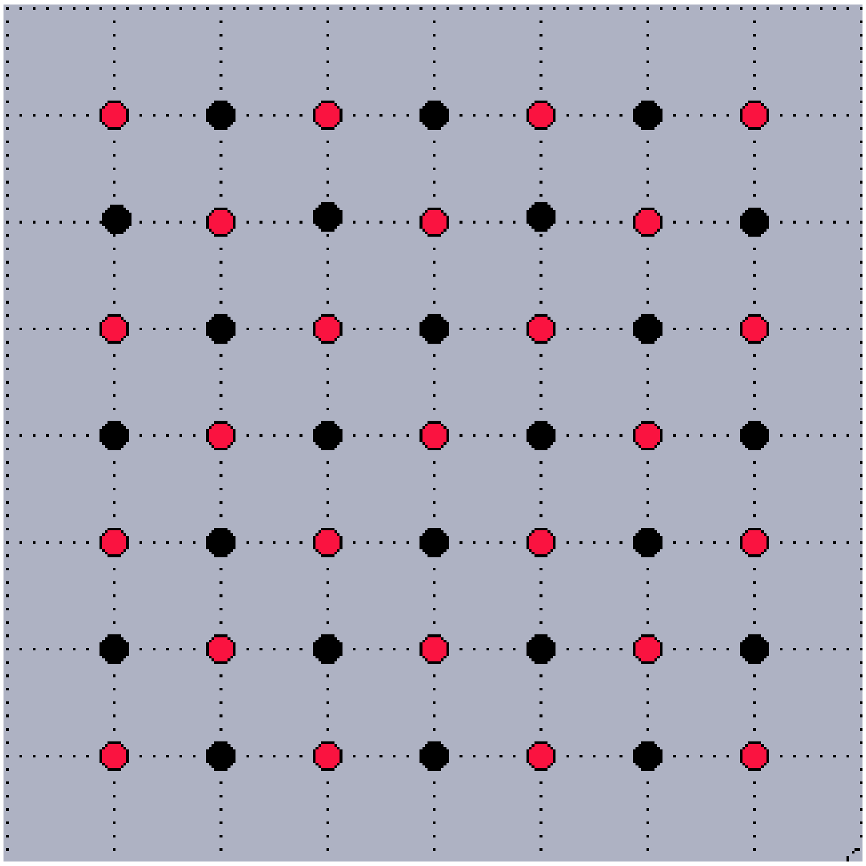

FIG. 4.1. *Red-black ordering of the unknowns for the red-black Gauss-Seidel algorithm. Here, the size of the grid is 9 and we have not shown the boundary nodes.*

data in memory for a machine with a cache-line size of 4 elements is shown in Fig. 4.2. We attempt to improve the performance of the standard algorithm by making it more cache aware.

We study four versions of a 2d red-black Gauss-Seidel with constant coefficients. The first is the standard algorithm, which updates all the red nodes in the grid and then updates all the black nodes in the grid per iteration.

The second and third versions are based on the pioneering work done by a group of researchers that include Craig Douglas, Ulrich Ruede, Christian Weiss, Jonathan Hu, Wolfgang Karl and Markus Kowarschik [1] [7]. They successfully optimized the above program to turn it into a cache aware program. In the standard version, a red update sweep through the whole grid is followed by a black update sweep through the whole grid. They improved the standard version by using the fusion technique to update red and black nodes in pairs. This technique fuses two consecutive sweeps through the grid, which update the red and black points separately, together to one sweep through the grid. This is the basis for our second version.

Our third version is based on the 2d blocking technique that they use. The fusing technique of the second version applies only to one single red-black Gauss-Seidel sweep. To perform several successive red-black Gauss-Seidel iterations, they use a 2d blocking tech-

ETNA
Kent State University
etna@mcs.kent.edu

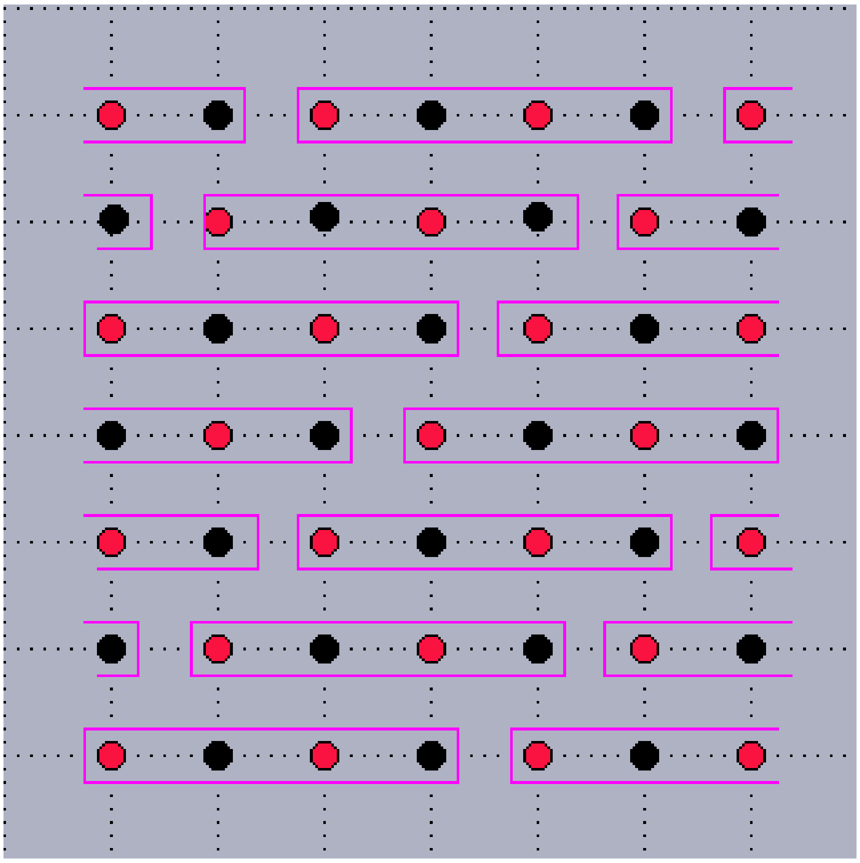

FIG. 4.2. *Standard layout of the red-black Gauss-Seidel unknowns in memory for a cache-line size of 4.*

nique. The key idea is to move a small two-dimensional block through the grid updating all the nodes within the block. The block is shaped as a parallelogram (Fig. 4.3) to obey all the data dependencies: a red can be updated for the i^{th} time if all its neighbouring blacks have been updated for the $(i-1)^{th}$ time; a black can be updated for the i^{th} time, if all its neighbouring reds have been updated for the i^{th} time. (We have showed that the best block shape for this operation is a parallelogram in our previous work [10]).The update operations within the parallelogram can be performed in a line-wise manner from top to bottom. For this, they work in diagonals in the parallelogram from top to bottom. As soon as the red points in the uppermost diagonal have been updated, the black points in the next diagonal within the parallelogram can also be updated. The updates continue in that manner. The maximum number of updates that are possible for one position of the parallelogram window respecting all data dependencies is done before the parallelogram is moved to the next position. At the boundaries, some boundary handling is done. This 2d blocking technique is implemented in our third version of the Gauss-Seidel.

The fourth version is our own contribution in which we try to further improve the performance of the third version by studying the access pattern of data and then creating a new layout for the data. It is seen that in the third version, the nodes in the grid are accessed diagonally. Thus, instead of numbering the matrix using rows and columns, we created a single array which stored the data diagonally. The key is to ensure that reds and blacks are stored

ETNA
Kent State University
etna@mcs.kent.edu

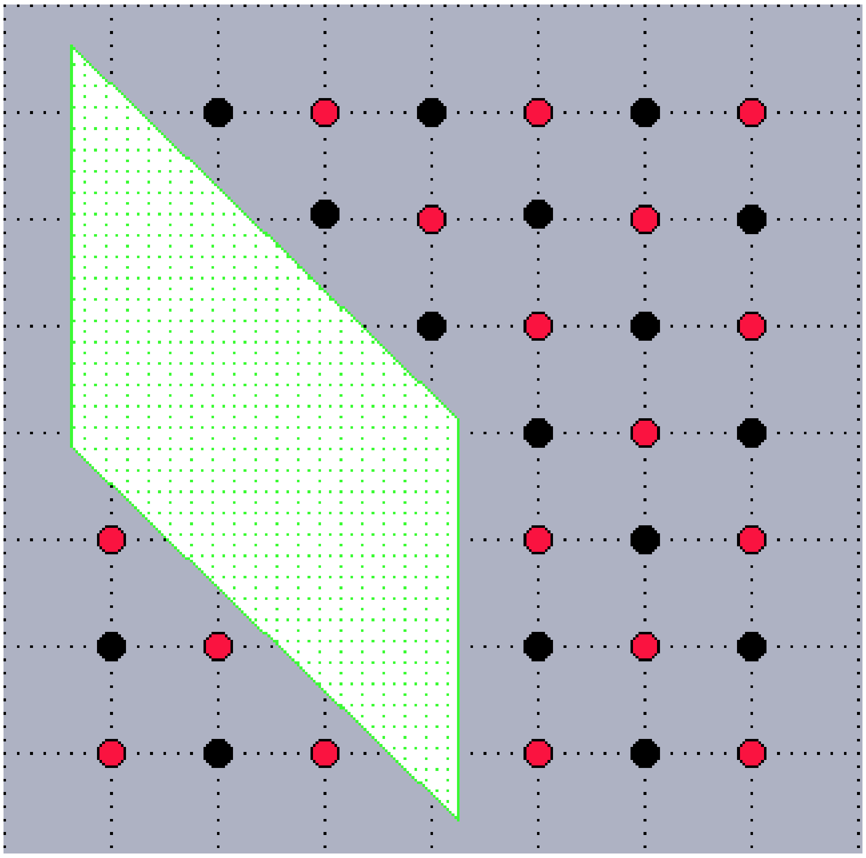

FIG. 4.3. *The parallelogram blocking technique for the red-black Gauss-Seidel*

in contiguous memory locations which is not true for row or column based ordering. For the actual implementation of the new data layout, we used a simple mapping. We inserted the mapping function to get the new array element for the (row,column) based node of the old code. This way, the implementation of the new layout is simple. It is just a matter of replacing the old data co-ordinates in code with the new mappings. The new layout of data in memory for a machine with a cache-line size of 4 is shown in Fig. 4.4.

5. Results. We performed numerous experiments on both SPARC based and Pentium based machines. The applications were implemented as different versions to reflect the standard, fused, blocked and data laid schemes. To ensure fairness, the different versions of an application had the same tasks in the code although the implementations differed. The different versions were also numerically correct as the answers from them were checked. The execution time was measured for the actual computational core. For each choice of parameters, several tests were done and the averages are listed in this paper.

We chose a parallelogram block size of 16×16 for our blocked and data laid implementations. Although not presented here, we experimented with different blocking sizes [10] and found that the execution time reduced as we enlarged the block size due to the more effective use of the cache. The best block size for the data laid version is theoretically limited by the L2 cache size. In our implementation however, in addition to the above restriction, we had the

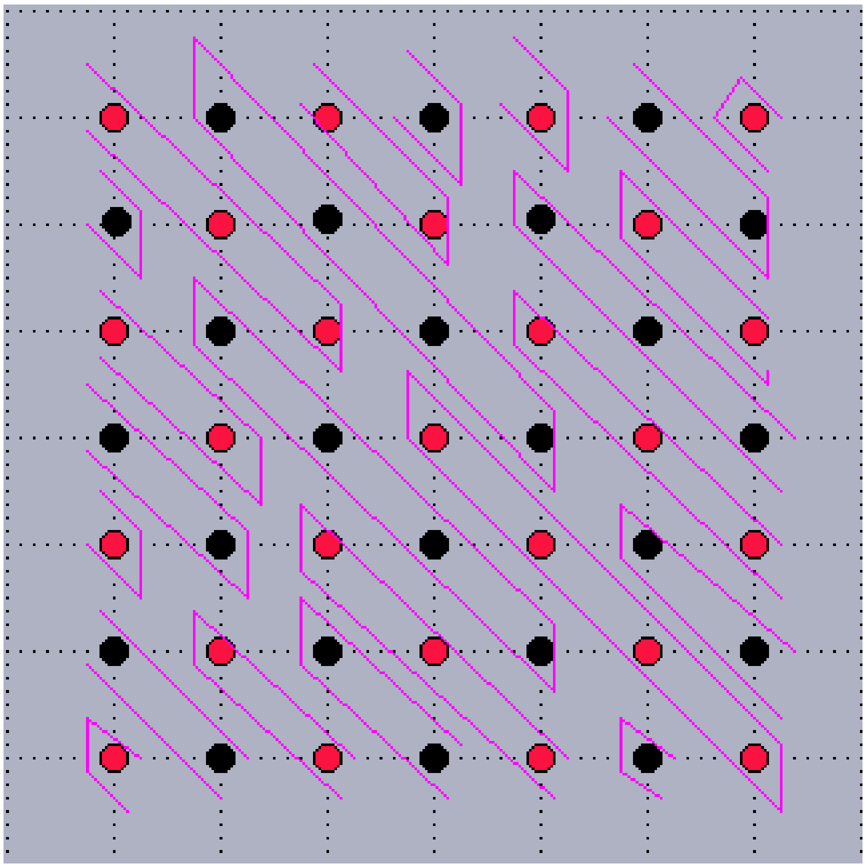

FIG. 4.4. *The new layout of data in memory for Gauss-Seidel using the cache aware data layout scheme. Here too, the size of a cache-line is 4.*

limitation for the maximum block height to be twice the number of Gauss-Seidel iterations, which was 16.

The tests that we report here were done on a Sun Ultra 10 (UltraSPARC-IIi 333MHz) machine. The machine had a memory of 128 Mbytes. It had a 16kB level 1 instruction cache and a 16kB level 1 data cache. This data cache is a direct mapped write-through cache. The machine had a unified level 2 cache of 2 MB which was writeback and direct mapped with a cache-line size of 64 bytes.

Table 5.1 gives the execution times for grids whose lengths were powers of two. As expected, the compiler optimizations (OST) bring easy performance improvements. Fused scheme (OFT) improves over the compiler optimized standard version in general. And the blocked scheme (OBT) improves over the fused scheme in general. For small grid sizes, we do not get an improvement from the data laid scheme since the mapping overheads outweigh the benefits at such small grid sizes. However, for large grid sizes that are not fully resident in the L2 cache (i.e., beyond 257×256) and that fit in memory, our data laid scheme gives significant performance improvements over the blocked versions. The best result is seen for the 1025×1024 scheme in which our scheme gives around 84% improvement over the blocked version. When the grid does not fit in memory, as when the grid size is 4097×4096, our scheme fails to give improvements over the blocked version. This we believe is due to

ETNA
Kent State University
etna@mcs.kent.edu

TABLE 5.1
Execution times for 2d Gauss-Seidel on the SPARC machine when the grid length is a power of two (UST - compilation-Unoptimized Standard Time; OST - compilation-Optimized Standard Time; OFT - compilation-Optimized Fused Time; OBT - compilation-Optimized 16×16 *Blocked Time; OLT - compilation-Optimized* 16×16 *data Laid Time; Imp - percentage improvement of the data laid version over the blocked version)*

Grid Size	UST (Sec)	OST (Sec)	OFT (Sec)	OBT (Sec)	OLT (Sec)	Imp
33×32	0.0012	0.0004	0.0004	0.0004	0.0005	-16.01
65×64	0.0066	0.0026	0.0020	0.0016	0.0020	-19.14
129×128	0.0239	0.0096	0.0080	0.0071	0.0081	-12.98
257×256	0.1060	0.0384	0.0318	0.0334	0.0324	3.09
513×512	0.4531	0.1905	0.1575	0.1822	0.1584	13.02
1025×1024	3.9358	2.9146	3.1577	3.7115	0.5885	84.14
2049×2048	11.3680	6.6382	6.2991	6.7230	2.4044	64.24
4097×4096	2218.271	2007.227	988.584	148.679	201.971	-26.39

TABLE 5.2
L1 Cache miss rates for the SPARC machine (OB - miss rate for the compilation-Optimized 16×16 *Blocked; OL - miss rate for the compilation-Optimized* 16×16 *data Laid; Imp - percentage miss rate improvement of the data laid version over the blocked version)*

Grid Size	OB	OL	Imp
33×32	0.70	0.43	38.57
65×64	3.16	1.25	60.44
129×128	6.02	1.86	69.10
257×256	10.20	2.42	76.27
513×512	15.75	4.80	69.52
1025×1024	61.19	18.49	69.78

the variable overheads that we use in our implementation turning out to be significant for out-of-memory grid sizes. Improving our scheme for out-of-memory applications is planned for our future work. We could achieve this by reducing the overheads in our scheme whose impact is seen especially for out-of-memory grid sizes.

We then performed profiling tests using a trace driven cache model developed by us at Uppsala University. The cache model simulated a direct mapped cache similar to the L1 cache of our Sun machine. The address traces generated by our Gauss-Seidel execution was fed into this profiler which in turn modelled the corresponding cache operation to give out cache miss rates. Results from profiling tests for the Gauss-Seidel on our Sun machine are given in Table 5.2. The results validate our execution time results and show that data laying reduces the cache miss rate.

The execution times for grids whose lengths were not powers of two are shown in Table 5.3. The results show the similar trends observed for the power of two case. Here again, our data laid scheme gives satisfactory improvements over the blocked version for large grid sizes.

To evaluate the portability of this technique, we performed tests on a Pentium II machine. This machine had a clock of 400 MHz and a level 1 data cache of 32kB and a level 2 data cache 512kB. The results are shown in Table 5.4. The results show that cache laying benefits are also applicable for this architecture. Since this machine had a memory larger than that of our Sun, the data laying result does not show a performance degradation for the 4097×4096 case. Infact, improvements show a promising increase with increasing grid size.

6. Conclusions. The results in this paper prove the benefits of cache aware computing and the added benefits that are available through cache friendly layout transformations. With simple cache friendly data layout modifications to an existing program, significant performance improvements are possible. Since the layouts are suitable for any present generation

TABLE 5.3
Execution times for 2d Gauss-Seidel on the SPARC machine when the grid length is not a power of two (UST - compilation-Unoptimized Standard Time; OST - compilation-Optimized Standard Time; OFT - compilation-Optimized Fused Time; OBT - compilation-Optimized 16×16 Blocked Time; OLT - compilation-Optimized 16×16 data Laid Time; Imp - percentage improvement of the data laid version over the blocked version)

Grid Size	UST (Sec)	OST (Sec)	OFT (Sec)	OBT (Sec)	OLT (Sec)	Imp
265×264	0.1120	0.0420	0.0350	0.0328	0.0353	-7.09
385×384	0.2330	0.0910	0.0750	0.0772	0.0760	1.55
521×520	0.4380	0.1830	0.1437	0.1310	0.1407	-6.87
769×768	1.0390	0.5135	0.5460	0.3627	0.3117	14.06
1033×1032	2.8733	1.4665	1.1870	0.6153	0.6283	-2.07
1537×1536	5.4525	2.8077	2.3040	2.1974	1.3086	40.45
2057×2056	10.1990	5.4860	4.4020	2.8940	2.2950	20.70
3073×3072	30.4550	14.0440	13.0600	12.1500	6.2470	48.58

TABLE 5.4
Execution times for 2d Gauss-Seidel on the Pentium machine when the grid length is a power of two (UST - compilation-Unoptimized Standard Time; OST - compilation-Optimized Standard Time; OBT - compilation-Optimized 16×16 Blocked Time; OLT - compilation-Optimized 16×16 data Laid Time; Imp - percentage improvement of the data laid version over the blocked version)

Grid Size	UST (Sec)	OST (Sec)	OBT (Sec)	OLT (Sec)	Imp
33×32	0.00148	0.00039	0.00059	0.00052	12.31
65×64	0.0067	0.0023	0.0026	0.0021	17.65
129×128	0.0273	0.0092	0.0120	0.0088	26.87
257×256	0.1299	0.0641	0.0519	0.0355	31.55
513×512	0.561	0.314	0.263	0.148	43.62
1025×1024	2.452	1.284	1.052	0.600	42.97
2049×2048	9.831	5.442	4.398	2.474	43.75
4097×4096	39.122	21.923	19.385	10.129	47.75

computer which promotes performance improvement through contiguous data storage, the technique is also highly portable. As processor speeds continue to increase relative to memory speeds, this technique should become even more important for future processors.

In conclusion, there are several steps that we recommend to application programmers to make their programs more cache aware. These steps are:

1. Understand the application.
2. Change the algorithm through established cache aware computing techniques like fusion, blocking, and loop interchange, so that it suits the cache of the hardware platform on which the program is to run.
3. Understand the current data access pattern of the algorithm and design new cache aware data layouts based on the access pattern.
4. Implement the new data layouts in code.
5. To further improve performance, use prefetching programming constructs (as and when and if they are developed by the programming language community) to prefetch the data before their use.
6. Check the different optimization parameters of the selected compiler, and compile the code with the best performance options for the target hardware platform.

7. Future Work. As an immediate extension to our data laying experiments, we like to direct our attention to reduce the overheads and to improve the performance of the red-black Gauss-Seidel for out-of-memory grid sizes. Also, we plan to develop mathematical models for the expected performance of the blocking and data laid strategies which would help programmers evaluate different programming solutions before selecting a suitor for implementation.

We have also applied the technique of data laying to a matrix multiplication application and have noticed significant performance gains [11]. However, our research up to now has only dealt with regular applications in which it was relatively easy for a programmer to transform an existing access pattern to a cache aware access pattern. Therefore our future work will also include extending this technique to irregular applications in addition to using it for other applications. We hope to investigate for example, variable coefficient 2d Gauss-Seidel, unstructured grids, 3d Gauss-Seidel and also applications with recursive data structures and indirect data. This work may result in a whole new approach to the design of data structures.

There are a few more directions that we hope to pursue. Checking the benefits of the techniques of data laying in parallel computing environments is one. Tools for programmers to make cache aware programs that reduce the memory barrier is another. And there is another promising direction in investigating complier techniques to automate data laying. Though beyond the scope of our area, investigating other applications that would benefit from cache aware work such as data base applications is also another direction of work.

Acknowledgments. The author thanks his Supervisor Prof. Richard Wait at Uppsala University, Sweden. He also thanks Prof. Kithsiri Samaranayake, Dr. Nihal Kodikara and Dr. Ruvan Weerasinghe of the University of Colombo, Sri Lanka.

REFERENCES

[1] C.C.DOUGLAS, J.HU, M.KOWARSCHIK, U.RUEDE AND C.WEISS, *Cache Optimization for Structured and Unstructured Grid Multigrid* , Electronic Transactions on Numerical Analysis, 10 (2000), pp. 21-40.

[2] W.HACKBUSCH, *Multigrid Methods and Applications*, Springer-Verlag, Berlin, 1985.

[3] J.L.HENNESSY AND D.A.PATTERSON, *Computer Architecture: A Quantitative Approach*, Second ed., Morgan Kauffman Publishers, San Mateo, California, 1996.

[4] G.RIVERA AND C.-W.TSENG, *Data Transformations for Eliminating Conflict Misses* , in Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), Montreal, Canada, June 1998.

[5] S.P.VANDERWIEL AND D.J.LILJA, *Data Prefetch Mechanisms*, in ACM Computing Surveys, Vol.32, No.2, June 2000.

[6] G.RIVERA AND C.-W.TSENG, *Tiling Optimizations for 3D Scientific Computations*, in Proceedings of the SC'00 Conference, Dallas, Texas, Nov.2000.

[7] C.WEISS, W.KARL, M.KOWARSCHIK AND U.RUEDE, *Memory Characteristics of Iterative Methods*, in Proceedings of the ACM/IEEE SC99 Conference, Portland, Oregon, Nov. 1999.

[8] U.RUEDE, *Technological Trends and their Impact on the Future of Supercomputers* , in High Performance Scientific and Engineering Computing, Proceedings of the International FORTWIHR Conference on HPSEC, H.-J. Bungartz, F.Durst, and C.Zenger, eds., vol. 8 of Lecture Notes in Computational Science and Engineering, Springer, Mar. 1998, pp. 459-471.

[9] S.CHATTERJEE, V.V.JAIN, A.R.LEBECK, S.MUNDHRA AND M.THOTTETHODI, *Nonlinear Array Layouts for Hierarchical Memory Systems* , in Proceedings of the 1999 International Conference on Supercomputing , May 1999.

[10] M.SILVA AND R.WAIT, *Study of Tile Shapes for a Cache Aware Iterative Method*, in Proceedings of the 2000 International Informational Technology Conference , Colombo, Sri Lanka, January 2001.

[11] M.SILVA, *Application Programmer Directed Data Prefetching* , Master's Thesis, York University, Toronto, Canada, July 2001.

[12] H.SAGAN, *Space-Filling Curves* , Springer-Verlag, 1994. ISBN 0-387-94265-3.