Optimizing Performance of Superscalar Codes For a Single Cray X1 MSP processor

Hongzhang Shan, Erich Strohmaier, Lenoid Oliker

Future Technology Group/Computational Research Center
Lawrence Berkeley Research Laboratory
One Cyclotron Road
Berkeley, CA 94720

{hshan, estrohmaier, loliker@lbl.gov}

Abstract

The growing gap between sustained and peak performance for full-scale complex scientific applications on conventional supercomputers is a major concern in high performance computing. The recently-released vector-based Cray X1 offers to bridge this gap for many demanding scientific applications. However, this unique architecture contains both data caches and multi-streaming processing units, and the optimal programming methodology is still under investigation. In this paper we investigate Cray X1 code optimization for a suite of computational kernels originally designed for superscalar processors. For our study, we select four applications from the SPLASH2 application suite (1-D FFT, Radix, Ocean, and Nbody), two kernels from the NAS benchmark suite (3-D FFT and CG), and a matrix-matrix multiplication kernel. Results show that for many cases, the addition of vectorization compiler directives results faster runtimes. However, to achieve a significant performance improvement via increased vector length, it is often necessary to restructure the program at the source level – sometimes leading to algorithmic level transformations. Additionally, memory bank conflicts may result in substantial performance losses. These conflicts can often be exacerbated when optimizing code for increased vector lengths, and must be explicitly minimized. Finally, we investigate the relationship of the X1 data caches on overall performance.

1. Introduction

The growing gap between sustained and peak performance for full-scale complex scientific applications on conventional supercomputers is a major concern in high performance computing. The problem will be exacerbated by the end of this decade, as mission-critical applications will have computational requirements that are at least two orders of magnitude larger than current levels. Superscalar architectures are unable to efficiently exploit the potentially large number of floating-point units that be can fabricated on a chip, due to the small granularity of their instructions and the correspondingly complex control structure necessary to support it. Vector technology, on the other hand, provides an efficient approach for controlling a large amount of computational resources provided sufficient regularity in the computational structure can be discovered. Vectors exploit these regularities in the computational structure to expedite uniform operations on independent data elements. In this work, we examine the recently-released Cray X1 vector architecture. The X1 is designed to combine traditional vector strengths with the generality and scalability features of superscalar cache-based parallel systems. However, this unique architecture contains both data caches and multi-streaming processing units, making the choice of programming methodology a challenging question for application programmers. In this paper we investigate the Cray X1 code optimization for a diverse suite of computational kernels, originally designed for superscalar processors.

The scientific high-performance computing community has accumulated a large body of numerical applications over the last decade specifically optimized for execution on superscalar-based processors. A critical question is whether these codes can achieve high-sustained performance on modern parallel vector architectures, and if so, how much programming effort is required. The performance of several microbenchmarks, kernels, and applications on the X1 architecture has been recently reported [8,9,10]; however, the vector optimization strategies and the associated programming effort have not been examined in great detail. Although vectorization techniques have previously been studied on previous generations of platforms [11,12], the lessons of yesterday do not necessarily apply to modern systems such as the X1. For example, new compiler and microarchitectural technology obviates the need for manual implemented data chaining transformations. More importantly, today's vector systems are more complex, with deep memory hierarchies and multiple levels of parallelism. Simply optimizing for maximum vector length without considering cache reuse – as was done in previous generations of vector architectures – may not be sufficient to optimize application performance on the X1. Our study examines seven important scientific kernels with a broad spectrum of computational requirements and memory access patterns.

The rest of the paper is organized as follows. Section 2 briefly describes the Cray X1 platform. The scientific kernels are then presented in Section 3. The performance optimization process is discussed in Section 4 and a summary of the results is shown in Section 5.

2. Cray X1

The basic building block for Cray X1 is the multi-streaming processor (MSP), which consists of four identical single-stream processors (SSP). Each SSP has two 64-bit vector units and one 2-way superscalar unit. The clock frequency for the vector units is 800MHz. Each vector unit is capable of one 64-bit floating-point add and one floating-point multiply operation each cycle. Thus the peak floating-point performance for a SSP is 3.2GFLOPs/s. Each SSP has 32 vector registers holding 64 double-precision words, allowing up to 512 outstanding memory requests to hide latency. Additionally, all vector operations are performed under a bit mask, allowing loop blocks with conditionals to compute without the need for scatter/gather operations. The scalar unit runs at 400MHz with two 16KB caches (instruction cache and data cache).

The four SSPs inside a MSP share a 2-way set associative 2MB data cache, a unique feature for vector architectures. The cache is needed because the memory bandwidth is not large enough to saturate the vector units. The peak memory bandwidth is 34.1 GB/s (i.e. 2.7 Bytes/Flop) while the cache bandwidth is much higher at 51.2GB/s. The cache here, as in superscalar systems, is mainly designed to exploit the temporal locality of scientific applications. An X1 node consists of four MSPs sharing a flat memory through 16 memory controllers (Mchips). Each Mchip is attached to a local memory bank (Mbank), for an aggregate of 200GB/s node bandwidth. To build large configurations, a modified 2D torus interconnect is implemented via specialized routing chips. Finally, the X1 is a globally addressable architecture, with specialized hardware support that allows processors to directly read/write remote memory address in an efficient way.

The X1 programming model is designed to hierarchically leverage parallelism. At the SSP level, vector instructions allow a large number of SIMD operations to execute in a pipeline fashion, thereby tolerating memory latency and allowing for high-sustained performance. MSP parallelism is achieved by distributing loop iterations across each of the four SSPs. The compiler must therefore generate both vectorizing and multistreaming instructions to effectively utilize the X1.

3. Applications

We select seven applications for our study, which are well tuned for execution on cache-based superscalar platforms. Four applications, 1-D FFT, Radix Sorting, Ocean, and Nbody are originally from the SPLASH2 suites. We rewrote them in the MPI programming model and tuned them on the Origin 2000 platform [1,2]. The CG and 3-D FFT are obtained from NAS parallel benchmark suite v2.3 [3]. The last one is the commonly used dense matrix-matrix multiplication.

NAS CG: is a solver for sparse linear systems using conjugate gradient method obtained from NAS parallel benchmark suite. The core operation is the sparse matrix vector multiplication. Its main memory operations are random access to a small data set and sequential access to a large data set at the same time.

NAS FFT is a 3-D fast-Fourier transform partial differential equation benchmark also obtained from NAS parallel benchmark suite. The bulk of the computational work consist of a set of discrete multi-dimensional FFTs, each accessing the same single large array multiple times, but with different strides each time.

Ocean simulates eddy currents in an ocean basin. A cuboidal ocean basin is simulated, using a discretized quasi-geostropic circulation model. The principle data structures are about 25 two-dimensional arrays holding discretized values of various functions associated with the model's equations. These grids are partitioned among the processes into square-like subgrids. The equation solver used is a W-cycle multigrid solver that uses red-black Gauss-Seidel iterations at each level of the multigrid hierarchy. In order to compute the value for a grid points, it needs the data from all its nearest neighbors at that grid level.

1-D FFT is a double precision complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm described in [4], which is optimized to minimize the inter-process communications. The *n*-point data set is arranged in the form of a $\sqrt{n} * \sqrt{n}$ matrix, and the matrix is partitioned among the processors in blocks of $\sqrt{n}/p$ continuous rows each. The whole FFT structure is as follows: (i) transpose matrix, (ii) perform 1-D FFTs individually on local rows of size $\sqrt{n}$ each, (iii) multiply the elements of the resulting complex matrix by the corresponded roots of unity, (iv) transpose matrix, (v) perform 1-D FFTs individually on local rows, (vi) transpose matrix.

Nbody simulates the interaction of a system of bodies in three dimensions over a number of time steps, using a hierarchical N-body method. There are three main stages, building an oct-tree to represent the distribution of the bodies, browsing the oct-tree and calculating the gravitational force, updating the body positions and velocities. At the beginning of each time step, a process has to exchange information with other processes to build the tree. Then it will browse the tree from the top for each body and compute the gravitational force along the path. The data accesses are scattered and involve many pointer-chasing operations. Force calculation is the most time-consuming stage.

Radix sorts a series of integer keys in ascending order using the radix algorithm. The radix size used determines the number of iterations. For each iteration, it computes the histograms first and then moves the keys according to the histograms. There are two main arrays working as source and destination alternatively. The source data are read sequentially while the destination data are written scattered.

Matrix-Matrix Multiplication (MM) computes the product of two matrices, which is a very common basic operation in scientific computations.

4. Performance

This section examined X1 performance in MSP mode. The data set sizes for NAS CG, NAS FFT, Ocean, Nbody, 1-D FFT, Radix, and MM are class B, class C, 2050*2050 matrices, 2 million bodies, 16 million data points, 256 million integers, and 2048*2048 matrices respectively. The performances of the original version and optimized version are shown in Fig. 1. Note that the original versions are currently being used on platforms with superscalar processors. We first examine whether all the loops in the original version can be properly vectorized and multi-streamed by the vector compiler automatically. If the compiler cannot identify the data independence between iterations, compiler directives are added where suitable to inform the compiler the code region is data independent. The version with the proper compiler directives is referred to as the directed version. For most of the experiments, simply adding compiler directives is not sufficient to deliver the optimal performance – since the compiler is unable to effectively exploit all of the available data-parallelism within the loops or functions. We therefore have to explicitly change the algorithms or data structures in order to assist the compiler in identifying data-parallel regions. The restructured code is called the optimized version.

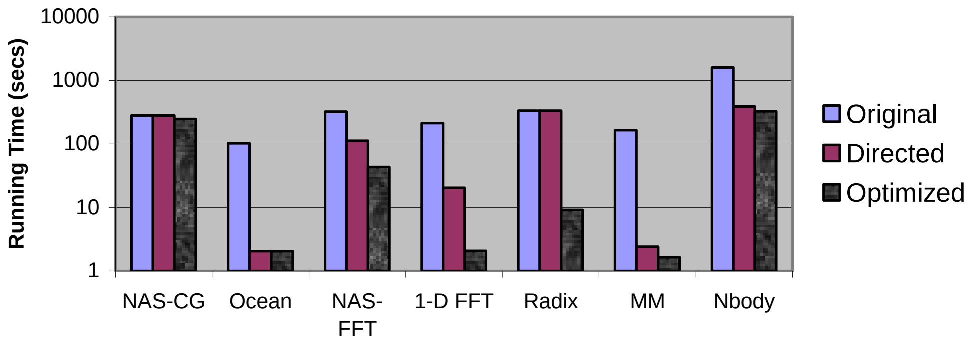

Fig. 1: The Performance of the Original, Directed, and Optimized Versions. The Y-axis is in Log-scale.

4.1 Effect of Using Compiler Directives

The compiler needs to make conservative assumptions about possible data dependencies to avoid potential race conditions. Compiler directives are therefore often necessary to allow for the effective vectorization and multistreaming of data independent regions. Figure 1 shows that compiler directives had almost no effect on the performance of CG and Radix. For CG, the main time-consuming component is the loops to compute the sparse-matrix vector multiplication. Here, the compiler was able to identify the data independence between iterations and vectorize and multistream the loops automatically. Therefore, the original version and the directed version deliver similar performance. The compiler directives have no effect on performance of radix either. Unlike CG, data dependencies within the loop iterations are preventing vectorization and multistreaming; causing the code to run on the scalar units. In other cases, the compiler directives can substantially improve the performance. For ocean, 1-D FFT, NAS FFT, and MM, the important loops can be both vectorized and multi-streamed. However, for Nbody, the loop can be multistreamed but not vectorized due to its code irregularity and complexity.

4.2 Application Restructuring and Performance Optimization

Adding compiler directives can exploit the data parallelism within the loops. However, in order to exploit the data parallelism across the loops or functions, the programs have to be restructured to generate more efficient execution codes. The average vector lengths of the directed version for NAS CG, ocean, NAS FFT, 1-D FFT, Radix, MM, and Nbody are 46.38, 63.15, 16.70, 9.8, 1, 3.71, and 64 respectively. For CG, further increase the average vector length without increasing data set sizes is difficult. The indirect memory access limits its performance. However, its performance can be improved about 10% if the inner loop is unrolled eight times explicitly (the optimized version). For Ocean, the average vector length has almost reached vector register length of 64 and no further optimization was necessary. We focus on increasing the vector length of other applications. For MM, we found that a naïve implementation using the stride access that will be intentionally avoided on superscalar processors delivers the best results on Cray X1.

NAS FFT There are two important implementation parameters in the NAS FFT, *fftblock* and *transblock*. The first parameter controls how many ffts are done at a time. The second parameter is the blocking factor for the transpose. The default values are 16 and 32 respectively, which are appropriate for most superscalar machines to maximize cache reuse. As suggested in the code, on the vector machines, the block size should be as large as possible, i.e. 256 for class B. The result using longer vectors is shown in Table 1 labeled as Vec-full. It only takes half of the time needed by the directed version. In the NPB 2.4 version, in order to reduce the amount of the memory required by the program, the time evolution array is no longer stored for all time steps but just for the first. With this efficient memory usage, the performance is further improved to 56.38 seconds as shown in Table 1. However, we find that if we set the *fftblock* and *transblock* value to 64 instead of 256, the running time becomes even shorter. The best running time we obtained is 43.14 seconds. By examining the hardware performance counters, we found that when the block size is 256, both the cache misses (E:2:0) and total requests to local memory (M:0:0) are two or three times higher than those when the block size is 64. This is related with the cache design. When the block size equals 64, all the data needed for the 64 ffts fits in the cache; when the block size becomes 256, the 2MB cache cannot hold all the data for these 256 ffts and causes more cache misses. Purely pursuing longer vector length without considering other factors may not lead to best performance results.

Original	Directed	Vec-full	No-evolve	Optimized
323	112	66.45	51.38	43.14

Table 1: The Running Times for NAS FFT (seconds).

1-D FFT The data in the 1-D FFT is organized in the form of a matrix. The local ffts of rows or columns are done one at a time. The average vector length for 16M double-complex data set is 9.8. Our first optimization is to perform the local ffts in a block manner as NAS FFT so that the ffts of 64 rows or columns can be done simultaneously. The average vector length now increases to 64. Surprisingly, the running time almost tripled from 20.4 seconds to about 70 seconds as shown in Table 2 labeled as Vec. One possible reason for the performance drop is the memory bank conflicts since the row length is a power of two. In order to verify our guess, we pad each row with additional space so that the row length is no longer a power of two. The padding result, labeled as optimized in Table 2, indicates that the memory bank conflict is indeed the reason for the performance loss. After vectorizing and padding, the optimized code now needs only 2.1 seconds for the whole run. Compared with the 20.4 seconds needed by the directed version, the running time now 10 time faster. We also find that the directed version can benefit significantly from padding: By simply padding the matrix for the directed version, the performance on the Cray X1 has been increased almost four fold (labeled as Padding).

Original	Directed	Vec	Padding	Optimized
213	20.4	69.8	5.8	2.1

Table 2: The Running Times for 1-D FFT (seconds).

Nbody Though in each time step of a simulation there are three phases, most of the running time is spent on the force calculation phase. More precisely, the running time is dominated by the loop to compute the forces. In this phase, each body traverses the oct-tree starting from the root. If the distance between the body and a visited node is large enough, the whole subtree rooted at it, will be approximated by that node. Otherwise, the body will visit all the children of the node, computing their effects individually and recursively. Therefore, the path through which a body traverses the oct-tree is dynamically decided and different from each other. Because of the complexity and irregularity of the loop body, we have not been able to reform the code to enable the compiler to vectorize this whole process. Multi-streaming the force calculation loop by compiler directives (the directed version) improves the performance of the original four times since all the four SSPs inside each MSP can now work simultaneously.

Original	Directed	Optimized
1600	388	326

Table 3: The Running Times for Nbody (seconds).

One way to optimize the code is to separate this process into two stages: finding all the nodes affecting the body first (browsing stage) and computing the force effects later (computing stage). In this way, we can easily vectorize the second stage and leave only the first stage not vectorized. The performance of the new version is shown in Table 3 labeled as optimized. The average vector length is increased from 3.71 to 27.40. However, the performance improvement is limited only to 20% compared with the directed version. By further analysis we find that the browsing stage is very time consuming and dominates the tree traversing process. For example, using the optimized version, the combined time of these two stages within a one-step simulation for 2 million bodies is 130 seconds. By removing the computing stage, the run time is only reduced to approximate 120 seconds. The pointer-chasing operation on the Cray X1 platform is quite expensive, since several instructions are need to form each 64-bit address.

Radix There are also three phases to sort the data using radix algorithm: browsing the local data to compute the local histogram, communicating among all the processors to compute global histogram, and reassigning the data based on the global histogram. However, none of these three phases in the directed version could be vectorized because of loop dependencies. For example, the code on the left side is used for the first phase to compute the local histogram stored in *bucket*:

```
For (I = 0; I < N; I++) {
      key_val = key_from[I] & bb;
      key_val = key_val >> shift;
      bucket[key_val]++;
}
```

```
For (j=0; j< VL; j++) {
  For (I=0; I < N / VL; I++) {
       key_val = key_from[j*N/VL+I] & bb;
       key_val = key_val >> shift;
       bucket_size[j*radix+key_val]++;
  }
}
```

N is the number of data. *Key_from* is the source data array. *Bb* and *shift* are used to compute which buckets the data belongs to. *Bucket* is used to record the local histogram. The computation of the value for the *bucket* prevents the loop from parallelized for vector units or efficiently multi-streamed on the Cray X1. In order to optimize the code, we used the “virtual processor” concept described in [5]. Here, each element of the vector register is viewed as a virtual processor. Each virtual processor is then assigned a portion of the data and a set of independent buckets so that it can work exactly as a processor in the optimized version. The corresponding code for the first phase is shown above (right).

Original	Directed	Vec-64	Optimized
333.5	333.5	151.1	9.16

Table 4: The Running Times for Radix (seconds).

VL is the number of virtual processors. The vector register size on Cray X1 is 64 elements and there are four SSPs in each MSP processor. Therefore, there are total 64*4 virtual processors available. The performance for the vectorized code is shown is Table 4 with label Vec-64. Now the average vector length changes from 1 to 64. But the performance only improved one time. The problem is also related to memory bank conflicts. Using 256 as the number of virtual processors causes significant memory bank conflicts and substantially hurts performance. Instead, if we use 63*4 as the number of virtual processors, the running time, labeled as optimized in Table 3, is almost 36 times better than the directed version.

Matrix-Matrix Multiplication The original version we used is a blocked matrix multiplication. On the cache-based superscalar platforms, block algorithms are often exploited to achieve better reuse of data in local memory. We select 256 as default block size on Cray X1. The running time for two dense matrixes with 2048*2048 double elements is around 2.4 seconds. The average vector length is 64. Compared with non-blocked code, blocked code is usually much more difficult to develop and understand. It also introduces an artificial parameter, block size, that has nothing to do with the algorithm and has to be tuned on each specific platform. Therefore, we like to examine how the non-blocked code works on the Cray X1 platform since it prefers longer vectors.

Original	Directed	Uni-stride	Optimized
164.31	2.42	5.73	1.64

Table 5: The Running Times for MM (seconds).

Suppose the matrix sizes for A and B are *M*N* and *N*K* respectively and the matrices are stored in one-dimensional arrays. There are two naïve ways to implement the matrix multiplication, stride and uni-stride:

Stride Implementation:

```
For (i=0; i < M; i ++) {
   For (j = 0; j < K; j++) {
      tmp = 0;
      For (k = 0; k < N; k++) {
         tmp += a[i*N+k] * b[k*K+j];
      }
      c[i*K+j] = tmp;
   }
}
```

Uni-stride Implementation:

```
For (i=0; i < M; i ++) {
    For (k = 0; k < N; k++) {
        For (j = 0; j < K; j++) {
            c[I*K+j] += a[i*N+k] * b[k*K+j];
        }
    }
}
```

The main difference between these two implementations is the order of the second and third loop. Both versions can achieve the same average vector length 64 on the Cray X1. And we expect the uni-stride version to deliver better performance due to its stride-1 memory access. Surprisingly, the stride version (labeled as optimized in Table 5) performs at least two times better than the uni-stride version (labeled as uni-stride in Table 5) and also better than the blocked version. If we unroll the most outer loop four times, the stride version can deliver even slightly better performance than the vendor-provided *dgemm* function in the *sci* library. By the performance tools on the Cray X1 (PAT), we find that there are actually no vector load references with stride bigger than 2. The main difference between these two versions is the number of load/store references. The uni-stride version has M times more store references and eight time more load references since each time a c element has changed, it has to stored first and read back later. By examining the assembler code, we find that the characteristics of stride access have been changed due to vector processing. Fig. 2 illustrates this effect.

In order to compute $c_{0,0}$, we need to load the first row of matrix A and the first column of matrix B. Accessing $a_{0,0} \sim a_{0,n-1}$ is continuous access. However, visiting $b_{0,0} \sim b_{0,n-1}$ is stride accesses. Because of the vector effect, $c_{0,0} \sim c_{0,VL-1}$ are computed simultaneously. Therefore $b_{i,0} \sim b_{i,VL-1}$ should also be accessed at the same time. The result is that the stride access is replaced by continuous access. We only need to load matrix A and C one time unlike the uni-stride version where the matrix C has to be accessed many times.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix} * \begin{pmatrix} \overbrace{b_{0,0} \quad b_{0,1}}^{VL} & \cdots & b_{0,k-1} \\ b_{1,0} \quad b_{1,1} & \cdots & b_{1,k-1} \\ \vdots \quad\quad \vdots & \vdots & \vdots \\ b_{n-1,0} \quad b_{n-1,1} & \cdots & b_{n-1,k-1} \end{pmatrix} = \begin{pmatrix} \overbrace{c_{0,0} \quad c_{0,1}}^{VL} & \cdots & c_{0,k-1} \\ c_{1,0} \quad c_{1,1} & \cdots & c_{1,k-1} \\ \vdots \quad\quad \vdots & \vdots & \vdots \\ c_{m-1,0} \quad c_{m-1,1} & \cdots & c_{m-1,k-1} \end{pmatrix}$$

Figure 2. Memory Access for Stride Matrix-Matrix Multiplication.

4.3 SSP vs. MSP

On the Cray X1, each MSP processor contains four identical SSPs that share a 2-way set associative 2MB cache. Programming in MSP mode is similar to developing codes on SMP-based platforms using the OpenMP model inside a SMP node and the MPI programming model across the nodes. But the compiler on the Cray X1 can automatically distribute loop iterations across each of the four SSPs and generate either SMP or MSP executable codes from the same source code. It does not require the programmer to provide explicit OpenMP directives so the programmer can freely select either MSP mode or SSP mode based on the application characteristics to achieve better performance without modifying the source code.

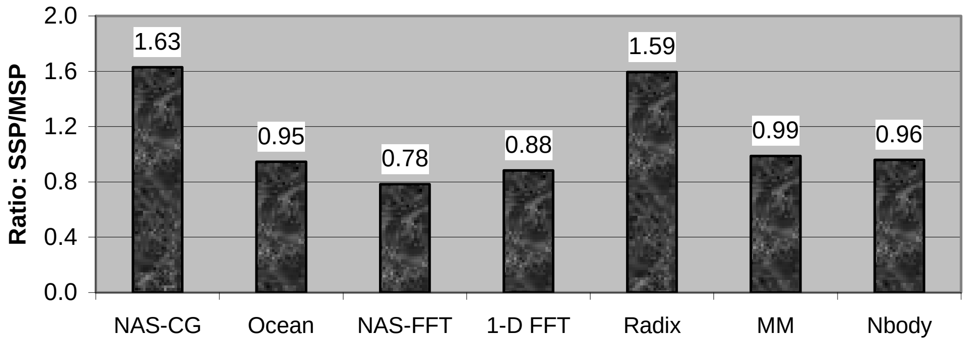

Figure 3: The Performance Ratio of one MSP vs. 4 SSP.

Fig. 3 presents the running time ratio of four SSP over one MSP using the optimized version we presented above. We find that for NAS-CG and radix, the SSP mode performs around 60% worse than the MSP mode. The main reason for the performance drop is that the SSP mode will increase the number of messages each processor has to process and reduce the size of each message. We expect further performance drop if more number of processors used. In other cases, its performance is better than that of the MSP.

4.4 Caching Effect

Compared with earlier vector architectures, the cache design is an innovative feature of Cray X1. Whether the cache has significantly effect on application performance on vector platforms is an interesting topic of study. On the Cray X1, each SSP has a 16KB scalar data cache (D-cache), while each MSP has a 2MB instruction and data cache (E-cache) that is shared by the 4 SSPs. The general cache and local memory access times are about 1x, 2x, and 7x for D-cache, E-cache and local memory respectively [6]. The advisory compiler directives `no_cache_alloc` can block cache line allocation for the specified data. However, this directive works only on objects that are vectorized. Table 6 shows the effect of using this compiler directive on the optimized version. For matrix-matrix multiplication, since the 2MB E-cache is too small, there is almost no temporal reuse. Therefore, the cache should have no substantial effect on its performance. Actually, using the *no_cache_alloc* directive slightly improves its performance. For Nbody, its performance degrades a little but not significant. Most of the Nbody code is not vectorized and thus should not be affected much either. For other applications, the performance is slowed 20 ~ 80%. The peak memory bandwidth is 34.1 GB/s, i.e. 2.7 Bytes/flop while the cache bandwidth is increase by 50% to 51.2GB/s. The cache is needed because the memory bandwidth is not large enough to saturate the vector units. Compared with the optimizations we have done above, such as increasing vector length and eliminating memory bank conflicts that may improve the performance dozens of times, the performance effect of cache is relatively smaller as we discussed in [7].

	NAS-CG	Ocean	NAS-FFT	1-D FFT	Radix	MM	Nbody
Cache	246.00	2.06	43.14	2.07	9.16	1.64	326.42
With no_cache_alloc	300.01	2.51	68.23	3.76	15.23	1.58	339.00
Slowdown	1.22	1.22	1.58	1.82	1.66	0.96	1.04

Table 6: The Cache Effect on Application Performance. The Slowdown is the Ratio of the Running Time with no_cache_alloc Directive vs. the Running Time without this Directive.

Finally, we end this performance section by showing the achieved performance of these applications on Cray X1 in terms of Mflops/s/MSP (shown in Table 7), a popular performance metric for scientific applications. The performance of radix is not displayed since it does not include floating-point operations. The peak performance of a MSP processor is 12.8Gflops/second. The achieved percentages of the peak performance vary widely for these applications, ranging from the 1.59% of Nbody to the 81.73% of matrix multiplication. The possibility of further performance optimization will be examined in future work.

	NAS-CG	Ocean	NAS-FFT	1-D FFT	MM	Nbody
Mflops/s	582	2830	2133	1651	10461	204
Percentage	4.55	22.11	16.66	12.90	81.73	1.59

Table 7: The Achieved Performance in terms of Mflops/s and Their Percentage of the Peak.

5. Summary

The Cray X1 is an innovative new architecture combining traditional vector feature with modern superscalar components. However, determining the optimal programming approach is still under investigation. In this work we examined seven important scientific kernels originally designed for superscalar platforms, with a broad spectrum of computational requirements and memory access patterns. Results show that the original unoptimized generally codes achieve low performance. In many cases, the addition of compiler directives was sufficient to substantially improve runtime. This was not true in some cases, however, where explicit application code restructuring was necessary to allow the compiler to effective exploit data-parallel code regions. Additionally, data reuse strategies were also utilized to maximize the cache efficacy. The code restructuring methodology was heavily dependent on the application details and largely ad hoc, maximizing vector length and cache reuse while avoiding bank conflicts. The overall performance effects of these optimizations were often quite substantial – leading to dramatic runtime improvements of 10X or even 100X over the original unmodified versions.

ACKNOWLEDGMENTS

The authors thank ORNL for providing access to the Cray X1 platforms. All authors are supported by the Office of Advanced Science Computing Research in the U.S. DOE Office of Science under contract DE-AC03-76SF00098.

References:

[1] Hongzhang Shan, Jaswinder Pal Singh, *Comparison of Message Passing, SHMEM and Cache-coherent Shared Address Space Programming Models on the SGI Origin 2000*, International Conference of Supercomputing, May 1999, Rhodes Island.

[2] Hongzhang Shan, Leonid Oliker, Rupak Biswas, Jaswinder Pal Singh, *Comparing Three Programming Models for Adaptive Applications on SGI Origin 2000*, Supercomputing, 2000, Dallas, Texas.

[3] NAS Parallel Benchmarks, *http://www.nas.nasa.gov/Software/NPB/*

[4] David H. Bailey, *FFTs in External or Hierarchical Memories*, Journal of Supercomputing, 4:23-25, 1990.

[5] Marco Zagha, Guy E. Blelloch, *Radix Sort For Vector Multiprocessors*, SC1991, Albuquerque, New Mexico.

[6] Optimizing Applications on the Cray X1 System, http://www.cray.com/craydoc

[7] Hongzhang Shan, Erich Strohmaier, "*Performance Characteristics of the Cray X1 and Their Implications for Application Performance Tuning*", ICS'04, Malo, France.

[8] T.H.Dunigan, M.R. Fahey, J.B.White, P.H. Worley, *Early Evaluation of the Cray X1*, SC2003, Phoenix, AZ. Nov. 2003.

[9] Leonid Oliker, Rupak Biswas, Julian Borrill, Andrew Canning, Jonathan Crater, M. Jahed Djomehri, Hongzhang Shan, and David Skinner, "A Performance Evaluation of the Cray X1 for Scientific

Applications”, 6th International Meeting on High-Performance Computing for Computational Science, Valencia, Spain, June 28-30, 2004.

[10] http://www.csm.ornl.gov/evaluation/PHOENIX/index.html.

[11] Wayne R. Cowell and Christopher P. Thompson, *Transforming FORTRAN DO loops to improve performance on vector architectures*, ACM Transactions on Mathematical Software, Vol.2 Issue 4, Pages 324-353, 1987.

[12] H. Cheng, *Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP*, IEEE Computers, 22(9):31-46, sep 1989.