Performance and Scalability Analysis of Cray X1 Vectorization and Multistreaming Optimization

Sadaf Alam and Jeffrey Vetter

Computer Science and Mathematics Division,
Oak Ridge National Laboratory
{alamsr, vetterjs}@ornl.gov

Abstract. Cray X1 Fortran and C/C++ compilers provide a number of loop transformations, notably vectorization and multistreaming, in order to exploit the multistreaming processor (MSP) hardware resources and its high memory bandwidth. A Cray X1 node is composed of four MSPs, which in turn are composed of four single streaming processors (SSP). Each SSP contains a superscalar processing unit and two vector processing units. Compiler vectorization provides loop level parallelization and uses the vector processing hardware. Multistreaming code generation by the compiler permits execution across the SSPs of an MSP on a block of code. In this paper, we analyze overall impact of loop-level compiler optimization on a scientific application called Parallel Ocean Program (POP). POP has been extensively optimized for X1 by instrumenting the code using X1 compiler directives. We compare and contrast automatic and manual optimization schemes available on X1 and analyze their impact on the code performance and scalability. Our results show that the addition of compiler directives increases the average vector length, thereby improving the single node performance significantly. However, this code scales at a slower rate as the local workload volume decreases and the communication costs increase.

1 Introduction

Modern vector computers like Japan's Earth Simulator and the Cray X1 combine the vector processing architecture and Massively Parallel Processing (MPP) in a single system design [5, 9]. They provide a very high memory bandwidth, which is key to realizing a high percentage of theoretical peak performance. The basic building block of the Cray X1 is the 12.8 GFlops multi-streaming processor (MSP) [1]. Each MSP is comprised of four single-streaming processors (SSPs). An SSP has two 32-stage 64-bit floating-point vector units and one 2-way superscalar unit. The SSP uses two clock frequencies, 800 MHz for the vector units and 400 MHz for the scalar unit. Each MSP has a MByte E-cache. Cray X1 systems implement Cray's new vector instruction set architecture, which support decoupled scalar and vector execution for maximum performance. The Cray X1 compiler automatically optimizes a loop for the eight vector units within an MSP [6]. It multistreams a long vectorized loop or unvectorized outer loop.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 304–312, 2005.

With aggressive compiler optimization flags, the Cray X1 compiler carries out a number loop transformations, which are presented in this paper.

We conducted experiments on a 512 MSP Cray X1 system at the Oak Ridge National Laboratory using a complete scientific application called Parallel Ocean Program. Several performance critical loops have been manually vectorized in the code [14] using compiler directives and dummy loop indices. We compare and contrast the code optimization strategies, in particular multistreaming and vectorization, and measure their impact on the overall performance of the application code. We also capture, using a hardware performance counter utility from Cray called pat_hwpc [7], detailed execution characteristics for complete application runs.

The layout of the paper is as follows: Section 2 presents the Cray X1 node architecture. Section 3 outlines various loop transformation schemes and compiler directives that exploit the MSP architecture. Section 4 provides a brief description of the scientific application. Section 5 details the experimental setup. Results are presented in section 6 and section 7 concludes the research.

2 Processing Node Architecture

The Cray X1 has a hierarchical processing node, memory and interconnection network design [1]. A processing node is composed of four multistreaming processors (MSPs), 16 GBytes of globally shared local memory, and an inter-MSP communication network. Local memory latency is the same for all processors within a single node and it is accessible by all processors on the node, processors on the other nodes and all I/O devices. Total peak local memory bandwidth for one node is 204.8 GBps, which also supports network traffic and I/O. Interconnect design and programming models for the X1 are described in [5].

Figure 1 shows the design blocks of an MSP. Each of the four SSP contain two vector units and one scalar unit. The X1 provides a large set of registers to reduce the number of memory accesses, reduce register spills, eliminate write-after-read dependencies and hide memory latency. The register set includes thirty-two 64-bit vector registers, 8 vector mask registers, 64 scalar registers, 64 address

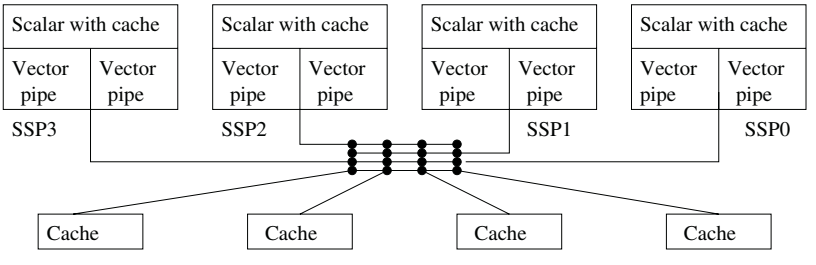

Fig. 1. Cray X1 multi-streaming processor architecture

registers, 8 control registers, a bit-matrix multiply register and a vector carry register [12].

There are three forms of cache in the X1 system. Each SSP has a 16KB scalar data cache and a 16 KB instruction cache, while each MSP has a 2MB instruction and data cache (E-cache) that is shared by the four SSPs within an MSP. Processors cache data from their local node modules only; references to memory on other node modules are not cached locally. In addition, the 32 KB of vector register space effectively functions as the lowest level of the processor data cache, similar to the L1 cache found on the microprocessor-based systems.

3 Vectorization and Multistreaming

Compiler vectorization provides loop-level parallelization of operations and uses the vector processing hardware: execution pipes, load buffers and functional unit groups. At the SSP level, vector instructions allow a large number of Single Instruction Multiple Data (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, which is referred as multistreaming. Vectorization and multistreaming can be intermixed, and multistreaming often extends outside the loop boundary. Figure 2 shows vectorization of a `do loop` with multistreaming. Individual iterations of a loop can be distributed over the four SSPs such that four 2-vector units of an SSP work together as a large single vector unit and can achieve an average vector length of 64.

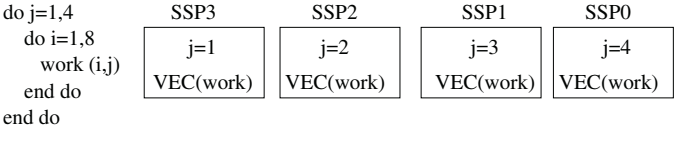

Fig. 2. Multistreaming and vectorization of a nested loop

3.1 Loop Transformations

Presently, compilers offer a number of loop transformations [2, 3], many by default, on almost all modern processor architectures. Similarly, there are a number of flags in the Cray compilers that allow for different levels of vector and multistreaming optimization [11]. For instance, `-O vector3` gives the compiler maximum freedom to vectorize loops, while the `-O stream3` allows for maximum multistreaming. Further, `-O aggress` option can optimize large loops. In addition, compiler directives can be embedded manually in the code. Here we briefly describe the Cray X1 compiler's loop transformations that exploit the multistreaming vector architecture. This includes loop interchange, collapsing, unwinding and loop fusion.

Loop Interchange. Compiler attempts to automatically interchange loops to maximize cache and vector register reuse and to reduce memory bandwidth usage. In the example below, the `i` elements of vector array `a` can be loaded before the `j` loop and `i` elements of vector array `a(i)` can be stored after the inner loop. After the loop interchange, the completely vectorized inner loop has no array `a` load and store.

```
before:                          after:
  do j = 1,m                       do i = 1,n
    do i = 1,n                       do j = 1,m
      a(i) = a(i)+b(i,j)               a(i) = a(i)+b(i,j)
    end do                           end do
  end do                           end do
```

Collapsing. The loop collapse schemes result in an increase in vector length and multistreaming and a reduction in loop overheads. Loop overheads are reduced by replacing a nested loop by a single loop.

Loop Fusion. In Fortran array syntax, every statement is considered a loop. Loop fusion minimizes loop overhead and maximizes register use. For example, array initialization `a=0; b=0` is fused and vectorized.

3.2 Code Instrumentation

Compiler directives are extensively used in the POP code [14]. Cray compilers offer a number of compiler directives for a range of compiler-directed optimization including vectorization and tasking directives, streaming directives and MSP optimization directives. For instance, a compiler directive can inform the compiler that a given loop can be collapsed (e.g. by inserting `!CDIR COLLPASE` before the loop). Likewise, other directives can indicate that it is safe to completely unroll, vectorize or inline a loop. Typically, during the code optimization process, a programmer generates the loopmark listing and identifies the optimized and unoptimized loops. A sample loopmark output is listed below:

```
ftn-6204 ftn: VECTOR File = file.f, Line = 625
  A loop starting at line 625 was vectorized.
ftn-6601 ftn: STREAM File = file.f, Line = 625
  A loop starting at line 625 was multi-streamed.
ftn-6004 ftn: SCALAR File = file.f, Line = 627
  A loop starting at line 627 was fused with the loop starting at
  line 625.
ftn-6289 ftn: VECTOR File = file.f, Line = 639
  A loop starting at line 639 was not vectorized because a
  recurrence was found on "TEMP" between lines 641 and 647.
```

Compiler directives can then be used, if due to some ambiguous data dependencies, the compiler is unable to transform or vectorize a loop. Given this

information the compiler can generate highly optimized code. In addition to compiler directives, dummy loops can be inserted. This is particularly useful for loop vectorization in Fortran code, which uses a verbose representation for array assignment operations.

4 Application

Cray X1 optimization strategies have been reported for a set of synthetic benchmarks [8]. In this paper, we conduct performance and scaling analysis of a scientific application called Parallel Ocean Program (POP), which was developed at the Los Alamos National Laboratory [13]. This code has been ported to a number of high-end supercomputing systems including two mainstream vector supercomputers: Japan's Earth Simulator and the Cray X1. A comparative performance study of POP on vector computers is presented in [4].

POP code is synchronous, and except for the infrequent I/O operations follows a Single Program Multiple Data (SPMD) programming paradigm. The code executes in a time-step fashion with the number of time steps specified in an input script file. There are two main processes in a POP time-step: baroclinic and barotropic. Baroclinic requires only point-to-point communication and is highly parallelizable. Barotropic contains a conjugate gradient solver, which requires global reduction operations. Moreover, the discretized POP grid is mapped and distributed evenly on a logical, two-dimensional processor grid [14].

Inter-processor communication in POP is performed via the Message Passing Interface (MPI) protocol. The exception is the conjugate gradient solver, which is implemented in Co-array Fortran [10] in order to reduce the communication requirement of this typically expensive component. Co-Array Fortran is a small set of extensions to Fortran 95 for SPMD parallel processing. It offers new set of rules for work distribution and data distribution within a program. POP optimization on Cray X1 are detailed in [14].

5 Experiments

For the performance evaluation experiments, the POP code is compiled with the options listed in table 1.

Additionally, using the `-rm` flag the compiler generates the loopmark listing, which identifies and explains optimization performed by the compiler on a given loop. The results reported in this paper are gathered from the loopmark files. In most cases a number of techniques identified in the paper have been applied to a single loop. For instance, a single loop can be collapsed, unrolled and vectorized by the compiler. Likewise, a compiler directive can result in a combination of loop transformations.

It is worth noting here that in a parallel code, a compiler hint or a directive should take into account the scaling effect of a given loop or loop indices. In an SPMD program like POP, often loop iterations and memory access operations

Table 1. Loop optimization applied to the POP application

Optimization	Description
default	equivalent to `-O2`
`-O3,aggress`	compiler aggressively optimize large loops
`-Ovector3,aggress`	aggressively vectorize loops
`-Ovector3,stream3`	maximum freedom to vectorize and multistream
Code instrumentation	as explained in [14]

are divided and distributed according to the size of the logical processor grid i.e. the two-dimensional MPI processor grid. The number of processor in `x` and `y` directions are specified at compile time in POP code. The number of simulated time steps in POP is controlled via `nstep` parameter. Scaling experiments are conducted for a fixed problem size, with a fixed `nstep` and by varying the number of processors in the `x` and `y` directions. Cray's `pat_hwpc` tool is used for calculating the runtime and average vector length.

6 Results

Table 2 shows the different loop optimization counts using the optimization listed in table 1. The table 2 also lists the runtime, percentage of vector instructions and average vector length, which has been collected by the pat_hwpc tool. The hand-coded version has a large number of single vector iterations and fused loops compared to the code that is generated by the Cray Fortran compiler without the compiler directives. As a result a larger fraction of instructions are executed by the vector processing units. An optimal utilization of the vector resources is translated in an increased average vector length. At the same time, compiler optimized code has a large number of unwounded loops compared to the hand-coded version. There is only a slight variation between different variants of compiler optimization.

Table 2. Number of individual loop transformations with different optimization schemes, pat_hwpc statistics and their impact on a single node performance

Optimization	Runtime (sec)	Average Vector Length	%Vector Instructions	Vector loops	Multi-streamed loops	Single Vector Iterations	Collapsed loops	Unwounded loops
default (`O2`)	254.32	37.243	93.06	1291	1243	76	34	192
`O3,aggress`	255.17	37.273	93.61	1329	1243	76	32	200
`vector3,aggress`	254.89	37.273	93.61	1329	1222	76	34	200
`vector3,stream3`	255.14	37.273	93.61	1329	1243	76	34	200
Instrumented	40.34	60.58	99.95	1306	1248	282	70	52

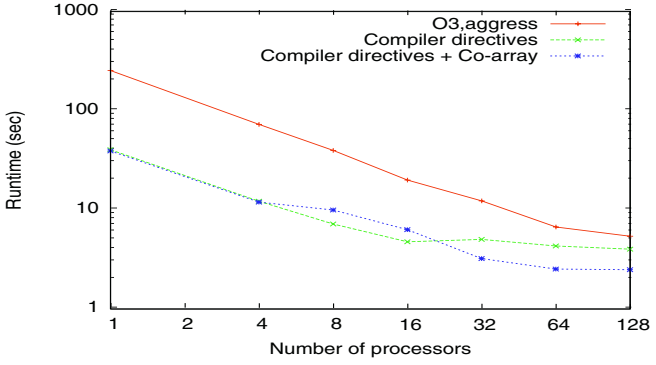

Fig. 3. Parallel efficiency

We also analyzed the scaling behavior of the code. For these fixed problem size experiments, increasing the number of processors did not significantly improve the performance in the instrumented code. Figure 3 shows the runtime values for the POP code generated: (1) with compiler-only optimization, (2) with compiler directives and (3) with compiler directives and co-array Fortran. We also compared the average vector length across different runs (shown in figure 4), which is an indirect measure of how effectively the vector resources are exploited. We note that the vector length first decreases with the decrease in local POP grid dimension and then stabilizes. We attribute this behavior to the vertical dimension `k` of the POP grid. (`i x j`) POP grid points scale with the underlying processor grid. The vertical `k` dimension is mapped across all processors and its value remains constant. That is why we do not observe a gradual drop in vector length by increasing the number of processors. Furthermore, the co-array Fortran optimization, which replaces the frequent MPI communication routines in the POP code result in the efficient execution and scaling of the code.

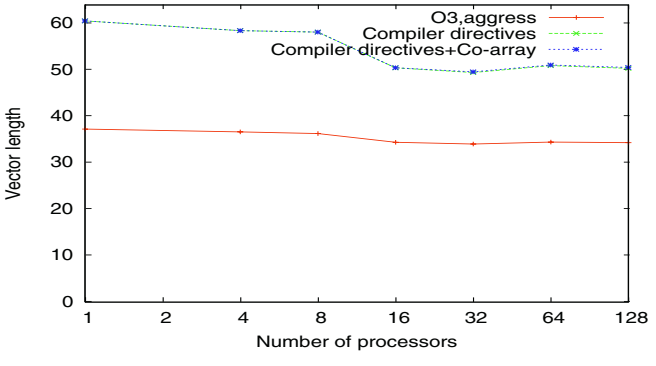

Fig. 4. Average vector length scaling

7 Conclusions

The Cray X1 design embodies advanced vector and superscalar concepts and it is highly optimized for floating-point intensive scientific calculations. We studied the loop vectorization and multistreaming strategies of the Cray X1 supercomputer and quantitatively evaluated their impact on the performance of a scientific application called POP. Furthermore, we analyze the scaling of single node loop optimization strategies, particularly code instrumentation strategies that involve adding compiler directives and dummy loop indices. We demonstrated that the single node performance of the instrumented code far exceeds that of the code which relies exclusively on the compiler for optimization, even when the most aggressive optimization flags are specified. At the same time however, for a fixed problem size, POP scales at a slower rate, as the local workload volume decreases. Hence, for larger problem sizes, the instrumented POP code is likely to more efficient than the code generated with aggressive compiler optimization.

Acknowledgements

This research was sponsored by the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

1. P. K. Agarwal, *et. al. ORNL Cray X1 evaluation status report*, Proceedings of the 46th Cray User Group Conference, 2004.
2. A. V. Aho, M. Hill and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Longman Publishing Co., Inc. USA, 1986.
3. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architecture: A Dependence Based Approach*, 1st ed. Morgan Kaufmann Publishers, 2001.
4. T. H. Dunigan, *et. al. Early evaluation of the Cray X1*, Proceedings of the 17th Annual International Conference on Supercomputing, 2003.
5. T. H. Dunigan, *et. al. Performance Evaluation of the Cray X1 Distributed Memory Architecture*, IEEE Micro 25(1), 2005.
6. *Optimizing Applications on the Cray X1 System: Loopmark Listings*. Available at http://www.cray.com
7. *Optimizing Applications on the Cray X1 System: Using CrayPAT Tools*. Available at http://www.cray.com
8. H. Shan and E. Strohmaier, *Performance Characteristics of the Cray X1 and Their Implications for Application Performance Tuning*, Proceedings of the 18th Annual International Conference on Supercomputing, 2004.
9. A. J. van der Steen and J. J. Dongarra, *Overview of Recent Supercomputers*, 2004.

10. R.W. Numrich and J.K. Reid, *Co-Array Fortran for Parallel Programming*, ACM SIGPLAN Fortran Forum, volume 17, no 2, 1998.
11. *Cray Fortran Compiler Commands and Directives Reference Manual*. Available at http://www.cray.com.
12. *Cray X1 System Overview*. Available at http://www.cray.com.
13. The Parallel Ocean Program Homepage, http://climate.lanl.gov/Models/POP
14. P. Worley and J. Levesque, *The Performance Evolution of the Parallel Ocean Program on the Cray X1*, Proceedings of the 46th Cray User Group Conference, 2004.