Compiler Technology for Blue Gene Systems

Stefan Kral, Markus Triska, and Christoph W. Ueberhuber

Institute for Analysis and Scientific Computing,
Vienna University of Technology,
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
mailto:skral@complang.tuwien.ac.at
http://www.math.tuwien.ac.at/ascot/

Abstract. Standard compilers are incapable of fully harnessing the enormous performance potential of Blue Gene systems. To reach the leading position in the Top500 supercomputing list, IBM had to put considerable effort into coding and tuning a limited range of low-level numerical kernel routines by hand. In this paper the Vienna MAP compiler is presented, which particularly targets signal transform codes ubiquitous in compute-intensive scientific applications. Compiling FFTW code, MAP reaches as much as 80% of the optimum performance of Blue Gene systems. In an application code MAP enabled a sustained performance of 60 Tflop/s to be reached on BlueGene/L.

1 Introduction

Blue Gene Servers. Top-performing supercomputers are usually based on the fastest processors available. In their latest hardware development, IBM went a radically different way, building Blue Gene servers [3] on an embedded-systems processor with low-power consumption, the IBM PowerPC 440.

To support scientific computing applications efficiently, IBM added a functional unit for double-precision scalar and 2-way SIMD floating-point arithmetic, extending the existing processor design by an auxiliary processor unit, yielding the PowerPC 440 FP2.

One node of a Blue Gene server comprises two PowerPC 440 FP2 processors (one dedicated to communication, the other one to computation), shared memory, and high-speed network interconnect hardware. The biggest installation built to date—BlueGene/L—is made up of the unprecedented number of 65,536 nodes integrated into a single distributed memory system. As of November 2005, Blue Gene servers take three out of the ten top positions on the Top500 supercomputing list, including the number one and two.

Automatic Performance Tuning Software. State-of-the-art numerical libraries in the field of linear algebra and signal processing are not based upon predetermined and fixed algorithms for performing the requested calculation, but utilize automatic performance tuning [4] to search the space of different algorithms and implementations for members of this set showing optimal run-time behavior. Rather than relying on formal performance models (covering

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 279–288, 2006.
© Springer-Verlag Berlin Heidelberg 2006

the utilization of the memory hierarchy, arithmetic operation count, instruction count, calling overhead of a procedure, and other relevant properties of the target architecture), they take actual runtime measurements obtained in numerical experiments to guide the process of automatic self-adaptation.

Automatic performance tuning systems often use automatically generated kernels that are long sequences of straight line code. For achieving high performance, these libraries heavily rely on the quality of the C compiler.

Experiments have uncovered a number of shortcomings of general purpose compilers when applying them to long, automatically generated straight line code, which opens up a performance gap between code generated by general purpose compilers and assembly code written by a skilled hand-coder.

The MAP Tool Chain. The Vienna MAP compiler tool chain aims at closing this performance gap, addressing domain-specific straight line codes produced by special-purpose program generators like `genfft` [8].

This paper describes a version of the MAP compiler targeting IBM's Blue Gene systems. The MAP compiler comprises a set of generic components arranged in the form of an open tool chain, communicating through a very narrow human-readable interface, which allows for (*i*) easy conservation of high-level information by means of annotation, (*ii*) introspection and injection of code, and (*iii*) easy experimentation with different arrangements of compilation stages.

Synopsis. Section 2 presents and discusses important properties of the target processor. Section 3 describes a 2-way single-instruction multiple-data (SIMD) vectorizer extracting parallelism out of basic blocks, Section 4 a versatile peephole optimizer for utilizing fused multiply-add (FMA) instructions, and Section 5 a Blue Gene specific backend that optimizes effective address calculations.

New contributions presented in this paper are improvements of (*i*) the vectorization method and of (*ii*) address-generation in the backend.

Section 6 demonstrates the impressive effects of the presented components and techniques on the performance of FFTW [9], the de-facto standard for the computation of discrete Fourier transforms (DFTs), running on Blue Gene servers.

2 The Blue Gene Processor

IBM's Blue Gene processor, the PowerPC 440 FP2, is a low-frequency (700 MHz) 32 bit processor with 32 integer registers, 32 SIMD floating-point registers, a short (7-stage) pipeline, large split L1 caches (32 kB for instructions, 32 kB for data), a fast non-pipelined multiplier, and support for 2-way super-scalar out-of-order execution. Integer registers are 32 bit, SIMD registers 128 bit wide.

Although the processor is a dual-issue design, not all conceivable pairs of instructions may be executed in parallel. Scalar arithmetic, SIMD data-reordering, and SIMD arithmetic use the same functional unit, and cannot be executed in parallel. At most one instruction per cycle may access naturally aligned memory.

The PowerPC 440 FP2 supports scalar [17] and 2-way SIMD [2] floating-point arithmetic, both operating on the same 2-way SIMD register file, with

scalar instructions working on the lower half of SIMD registers. Floating-point addition, subtraction, and multiplication are all fully pipelined, and available in double precision only. Support for single-precision floating-point data is offered for data loads/stores and by explicit rounding operations. Both for the scalar and the SIMD case, FMAs are available, which doubles the peak performance and improves the accuracy of the results by avoiding intermediate rounding.

FP2 offers a huge collection of vertical (inter-operand style) SIMD FMAs, including instructions that (*i*) perform different operations on different parts of the SIMD registers (e. g., addsub), (*ii*) use one part of a register as input for both operations, and (*iii*) combine a swap with an arithmetic operation.

Native support for horizontal (intra-operand style) SIMD is, however, completely missing in FP2. Emulating horizontal SIMD operations with a sequence of vertical and data reordering operations is considerably more expensive than on other SIMD ISAs (Tables 1 and 2).

Table 1. Instruction Count for Horizontal (H) and Vertical (V) Addition and Subtraction Operations. Uniform instructions perform two additions or two subtractions, while mixed instructions perform an addition and a subtraction.

op	3DNow!	Ext. 3DNow!	SSE2	SSE3	IA64	**FP2**
H / uniform	1	1	3	1	3	5
H / mixed	2	1	4	2	3	5
V / uniform	1	1	1	1	1	1
V / mixed	2	2	2	1	1	1

Table 2. Instruction Count for Data Reordering Operations. Uniform unpacks (unpackXX) combine the lower parts of two registers, while mixed unpacks (unpackXY) combine the lower part of one register with the upper part of another.

op	3DNow!	Ext. 3DNow!	SSE2	SSE3	IA64	**FP2**
unpackXX	1	1	1	1	1	2
unpackXY	2	2	2	2	1	2

Scalar computation can only be done in the lower half of the registers and some data may need to be moved. Mixing scalar and SIMD code is possible, but not at uniform cost.

The application binary interface (ABI) used in the Blue Gene environment [12] defines approximately half the registers as callee-saved, which can be a considerable disadvantage for small leaf procedures.

The PowerPC 440 FP2 lacks two important features for calculating effective addresses efficiently. First, the PowerPC ISA does not offer a combined *shift by a constant and add* instruction. Second, FP2 does not support *register+immediate* forms [5] for SIMD loads/stores.

3 The MAP Vectorizer for Blue Gene

Unlike SIMD-style vector computers, SIMD floating-point ISA extensions on general purpose processors operate on very short vectors. As this allows expressing parallelism on a very low level, not only loop-based vectorization techniques [19], but also more fine-grained ones, that extract the parallelism already present within a basic block, can be utilized.

To get the highest possible performance, a basic block vectorizer tries to maximally cover a scalar DAG with SIMD instructions natively supported by the target machine.

While our approach has some similarity to existing work like [6, 13, 14], our work is biased towards different assumptions about the class of input codes and about the target hardware. (*i*) As linear transform codes are highly structured, any divide-and-conquer based vectorization approach incurs high costs when connecting vectorized sub-graphs. (*ii*) Unlike SIMD ISAs on some DSPs, SIMD ISAs present on general purpose microprocessors do not allow scalar and SIMD to be mixed efficiently. (*iii*) As numerical kernels used in automatic performance tuning systems can be very large, finding a compromise between vectorization runtime and code quality is a key issue. (*iv*) Accesses to interleaved complex numbers naturally translate to 2-way SIMD memory instructions, which massively prunes the search space. However, for some kernels that do not have this kind of access, e. g., real FFTs, the vectorizer has to consider all combinations of all possible pairs of DAG inputs and DAG outputs.

Vectorization consists of two major steps, that are alternated until either the scalar DAG is covered with SIMD instructions or failure is discovered.

First, the vectorizer combines pairs of scalar variables to SIMD variables, ensuring that no scalar variable occurs in two SIMD variables and that the producers of the respective variables may be joined into a (pseudo) SIMD instruction.

Second, as the vectorizer combines two scalar instructions to one SIMD instruction, it propagates the layout requirements of the inputs and outputs of the newly extracted SIMD instruction, triggering the creation of new pairs.

Non-deterministic choice in this search process is handled by using depth-first search with chronological backtracking.

In an attempt to prune the search tree, the vectorization engine tries to detect failure branches early, allowing to traverse a much smaller part of the search space without missing any relevant part.

To further restrict the search space, pairs of scalar variables that can not occur as part of any solution are filtered out before vectorization is started.

The scalar DAG traversal order can have a profound impact both on the solution order and on the vectorization runtime. Earlier versions of the vectorizer [7, 15] always started at the outputs of the DAG, i. e., store instructions, traversing the DAG in a bottom-up fashion.

To improve on this, we added a top-down traversal style and borrowed the concept of domain variables [18] from constraint programming (CP). Domain

variables allow the vectorizer to keep track of all pairs of scalar variables that may be formed in the future. When traversing the scalar graph, the scalar variable that occurs in the smallest number of pairs, is picked as the next node to be visited (first-fail principle).

The combination of these traversal methods allows finding the optimal vectorization even for relatively large codes that use exclusively real arithmetic—a class of codes notoriously hard to vectorize.

Vectorization may yield more than one solution. A branch-and-bound based method is used to gradually find better and better solutions, until either optimality is proven or a time limit is reached. Generally, finding an optimal solution takes much less time than proving its optimality.

While all previous prototypes of the vectorizer have been specifically adapted to exactly one target architecture, the new version uniformly supports all target architectures taking target-specific data (as presented in Table 1) as input.

4 The MAP Optimizer for Blue Gene

The MAP optimizer only focuses on improving local structures (peepholes), rewriting sequences of instructions logically connected by data dependencies. Because of the locality of the approach, the global structure of the code, determined by the vectorizer, remains largely unchanged.

Implemented as a committed-choice term rewriting system, the optimizer is based on one or more sets of rewriting rules, each with a different priority. Out of all applicable rules, the rewriting engine picks one with the highest priority, and uses it to substitute instructions within a peephole with a semantically equivalent sequence of instructions. If no rule is applicable, a fixed-point is reached and the optimization terminates.

The optimizer uses two kinds of rules working in synergy. (*i*) *Improving rules* aim at an immediate improvement in code quality. Examples include rules for fusing two neighboring instructions into one, or rules handling horizontal SIMD instructions with neighboring SIMD swaps. (*ii*) *Assisting rules* do not immediate improve the code, but rather adapt the DAG such that improving rules may be applied. Examples include rules moving SIMD swaps or multiplications by constants within the DAG.

Apart from commonplace compiler optimizations [16], the optimizer tries to (*i*) shorten path lengths within peepholes, (*ii*) reduce the number of source operands by identifying domain-specific code patterns (e. g., the butterfly-ish code patterns typically occuring in FFT codes), and (*iii*) reduce the total number of instructions, both by eliminating superfluous instructions and by hiding some instructions in other ones, in particular by utilizing variants of SIMD FMAs.

5 The MAP Backend for Blue Gene

Unlike the two previously presented components, the MAP backend consists of a relatively large number of parts.

5.1 Effective Address Generation

All integer instructions in the code produced by the MAP compiler are devoted to either fulfilling the ABI calling convention or to calculating effective addresses.

While the code for fulfilling the calling convention has a constant size (regardless of the actual size of the procedure to be compiled) for procedure prolog and epilog, the code for the calculation of effective addresses may grow linearly with the number of memory accesses in the procedure to be compiled.

Experiments have shown that the portion of the code needed for the calculation of effective addresses often has a significant negative performance impact in case of algorithms having a high ratio of the memory access count compared to the number of arithmetic operations. All fast algorithms for linear signal transforms possess this property.

The IBM PowerPC 440 FP2 processor has DSP-like addressing mode limitations for SIMD loads/stores, minimizing the number of integer auxiliary instructions in of particular importance.

Basic Ideas. The calculation of effective addresses of elements of variably strided arrays (the actual stride is not known at compile time) can be done straightforwardly by using integer multiplication instructions. However, these instructions are expensive (low throughput, high latency) on all general purpose processors, including the IBM PowerPC 440 FP2.

The common approach to addressing this problem is *strength reduction*, which replaces complex instructions with sequences of simpler (high throughput, low latency) instructions like integer additions, subtractions, and shifts.

Implemented Solution. Doing strength reduction in a hard-coded fashion implies making instruction selection decisions without properly considering the temporal context, thereby missing opportunities to (*i*) reuse already calculated factors still residing in the register file and (*ii*) pick factors that could be beneficial for some proximate address calculation to be carried out in the near future.

To produce high-quality code, the MAP backend interleaves integer instruction selection and integer register allocation, thus removing a classical compiler optimization barrier.

Premature commitment to one particular factorization or reduction is avoided by utilizing a blended mixture of well-established search methods, depth-first iterative deepening (DFID) and dynamic programming (DP).

As exhaustive search for an optimal solution may not be possible for all but the smallest codes, the backend (*i*) looks at reasonably sized sub-problems, (*ii*) solves these sub-problems optimally, and (*iii*) combines the respective optima to one solution of the original problem. The quality of this solution depends on the amount of overlap of the sub-problems considered and on the size of these sub-problems.

To control the amount of search performed, the backend offers a set of parameters to directly control the speed and quality of the search, allowing to

trade compilation time for code quality, by specifying the size and the amount of overlap of the sub-problems.

5.2 Register Allocation

The MAP backend performs register allocation for all register files holding non-integer data in one pass, using the farthest-first policy [1, 10].

5.3 Scheduling

The MAP backend implements a set of various schedulers, covering a wide range from domain-specific high level scheduling to target-processor specific low level code reordering.

High Level. Two high level schedulers are part of the MAP backend. Both of them aim at a minimization of the register pressure.

The first high level scheduler implements an FFT specific topological sort of the computation DAG, attempting to enhance locality by minimizing variable life-span. This scheduler is directly derived from the scheduler of `genfft`, the program generator of FFTW.

The second high level scheduler performs local code reordering, trying to further reduce the register pressure for codes exhibiting a non-regular structure, e. g., SIMD-vectorized FFT codes.

Medium Level. The medium level scheduler reorders instructions taking latencies into account, thereby increasing the register pressure. By avoiding all dispensable movement, it preserves the original instruction order—obtained by high level scheduling—as much as possible.

Low Level. The low level scheduler specifically addresses execution properties of the target processor, implementing a list-scheduling algorithm that provides a runtime estimation of a given basic block. This scheduler is based on an in-order, super-scalar execution model of the target processor and handles both pipelined and non-pipelined instructions (like integer multiplication) well.

Execution models incorporate information about (*i*) instruction latencies, (*ii*) instruction throughput, (*iii*) issuing and decoding constraints, (*iv*) the mapping of instructions to functional units, and (*v*) register forwarding features.

6 Performance Results

To assess the performance impact of the presented techniques on Blue Gene systems, we compiled the compute-intensive numerical kernels of FFTW 2.1.5 with the following setups. *xlc_scalar* uses the XL C compiler without automatic vectorization. *xlc_vect* uses XL C with automatic vectorization. *xlc_mapvect* uses the MAP vectorizer and optimizer, producing C code with SIMD intrinsics compiled by XL C. *map_vect* uses the MAP vectorizer, optimizer, and backend.

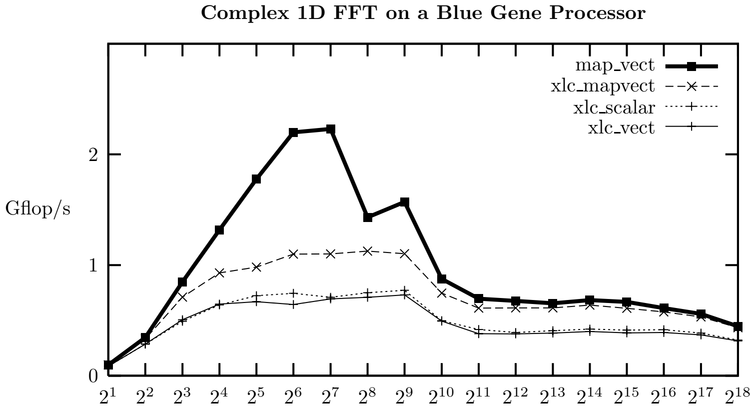

Fig. 1. Performance of Power-of-two 1D FFTs on the IBM PowerPC 440 FP2

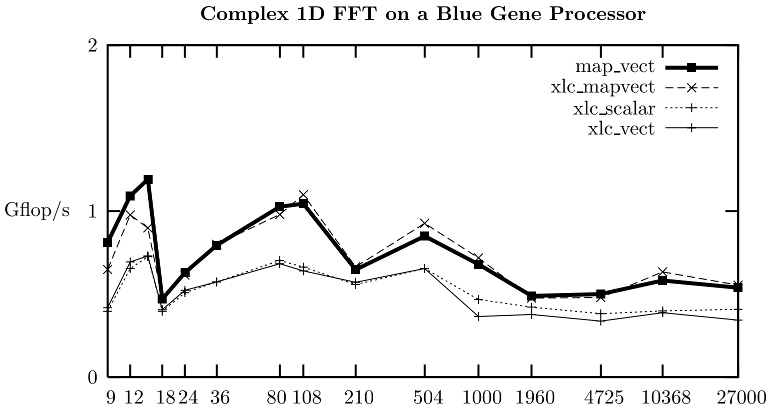

Fig. 2. Performance of Non-power-of-two 1D FFTs on the IBM PowerPC 440 FP2

Figs. 1 and 2 show the single-processor FFT performance achieved on Blue Gene systems by using various compilers and settings. All performance data are displayed in pseudo-Gflop/s, i. e., $5N \log N/T$.

Performance of Power-of-two Sizes. With very short vectors, calling the FFTW framework dominates the total cost. For medium sizes (2^3 to 2^9), all data fits into L1 cache, and the performance peaks—the MAP generated code for length 2^7 has 2230 pseudo Mflop/s, as opposed to 709 pseudo Mflop/s of the XL C compiled code. For transform lengths bigger than 2^{10}, data no longer fits into L1 cache, and the performance falls sharply.

Performance of Non-power-of-two Sizes. The performance shown in Fig. 2 is much more uneven than in the power-of-two case, because the chosen vector lengths have a larger number of different factors, leading to the use of many relatively small routines.

Effect of the Backend. We have examined the performance attributed to the compiler backend used (*xlc_mapvect* vs. *map_vect*), finding that the MAP backend produces much better code for compilation units consisting of one large basic block, while XL C profits from being able to perform its optimizations on units larger than one basic block, e. g., by loop unrolling.

It is noticeable that the backend does not give a significant performance gain in the non-power-of-two case (Fig. 2). This is due to the fact that FFTW normally does not include large kernels for non-power-of-two sizes as base cases. A comparable performance level as in the power-of-two case could be obtained if large non-power-of-two kernels were included into the library.

Instruction Count. For all codes investigated, the MAP vectorizer and optimizer for Blue Gene significantly reduced the instruction count by utilizing FP2 SIMD instructions. While the biggest part of the gain can be attributed to vectorization, the optimizer also has its share in code quality, by utilizing FP2 specific instructions, eliminating many SIMD swaps and multiplications.

For SIMD codes, the address generation part of the backend improves the code quality, by minimizing the number of integer instructions.

As FFTW kernels can be very large, minimizing the instruction count helps avoid hitting L1 instruction cache capacity limits.

Superior Performance Level. In the best cases, code produced by the MAP compiler runs at 80% of the performance that the best algorithm known in the literature could theoretically achieve on the target hardware.

MAP-compiled FFTW codelets enabled the material science code Qbox [11] to run with a sustained performance of 60 Tflop/s on BlueGene/L, thus reaching the second highest performance ever achieved by an application code.

7 Conclusion

The MAP compiler tool chain covers *all* stages of compilation that are important for achieving high performance in numerical software for linear signal processing transforms.

First, the code produced by a special purpose program generator, like FFTW's `genfft`, is vectorized, seeking an optimal utilization of the 2-way SIMD floating-point unit of IBM's PowerPC 440 FP2 processors.

Next, the MAP optimizer tries to minimize SIMD data reordering overhead and maximize utilization of FMAs and other FP2 specific idioms.

Finally, the code is compiled down to assembly, using (*i*) an optimal algorithm for register allocation for basic blocks, (*ii*) several levels of scheduling, and (*iii*) a clever instruction selection method for dealing with effective address generation on a processor with DSP-like addressing mode restrictions.

Performance data gathered in experiments with FFTW by itself and in the context of large application codes demonstrate the impressive performance—up to 60 Tflop/s—to be obtained by using the MAP compiler tool chain.

References

1. L. A. Belady. A study of replacement algorithms for virtual storage computers. IBM Systems Journal, 5(2):78101, July 1966.
2. K. Dockser. Oedipus Architecture: Extensions to PowerPC BookE for Hummer2. Technical report, IBM, August 2001.
3. J. E. Moreira et al. Blue Gene/L Programming and Operating Environment. IBM Journal for Research and Development, 49(2/3), 2005.
4. M. Puschel et al. SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, 93(2):232275, 2005.
5. S. Chatterjee et al. Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L. IBM Journal for Research and Development, 49(2/3), 2005.
6. R. J. Fisher and H. G. Dietz. Compiling for SIMD Within A Register. In Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC), pages 290304, 1998.
7. F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber. Efficient Utilization of SIMD Extensions. IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation, 93(2), 2005.
8. M. Frigo. A Fast Fourier Transform Compiler. Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 34(5):169180, May 1999.
9. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In Proceedings of the IEEE Intl. Conference on Acoustics, Speech, and Signal Processing, volume 3, pages 13811384. IEEE, 1998.
10. J. Guo, M. Garzaran, and D. Padua. The power of Beladys algorithm in register allocation for long basic blocks. In LNCS on Languages and Compilers for Parallel Computing, volume 2958, pages 374390. Springer-Verlag, 2004.
11. F. Gygi, E. Draeger, B. R. de Supinski, R. K. Yates, F. Franchetti, S. Kral, J. Lorenz, C. W. Ueberhuber, J. Gunnels, and J. Sexton. Large-Scale First- Principles Molecular Dynamics Simulations on the BlueGene/L Platform using the Qbox Code. In Proceedings of the ACM/IEEE Conference on Supercomputing, 2005. Gordon Bell Prize runner-up.
12. S. Hoxey, F. Karim, B. Hay, and H. Warren (editors). The PowerPC Compiler Writers Guide. Warthman Associates, 1996.
13. S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. ACM SIGPLAN Notices, 35(5):145156, 2000.
14. R. Leupers and S. Bashford. Graph-based code selection techniques for embedded processors. ACM Trans. Design Autom. Electron. Syst., 5(4):794814, 2000.
15. J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber. Vectorization techniques for the Blue Gene/L double FPU. IBM Journal for Research and Development, 49(2/3), 2005.
16. S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.
17. E. Sikha and R. Simpson. The PowerPC Architecture: A Specification for a New Family of RISC Processors. Morgan Kaufmann, 2nd edition, 1995.
18. P. van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, 1989.
19. H. Zima and B. Chapman. Supercompilers for Parallel and Vector Computers. ACM Press, New York, 1991.