Vectorization on Intel Xeon Phi: A Survey
Approach

Niraj J. Tiwari
Student
Department of Computer Engineering
PICT, Pune

Pujashree S. Vidap
Professor
Department of Computer Engineering
PICT, Pune

Abstract

In computer science, vectorization is the process of converting an algorithm from a scalar implementation to a vector process. It does an operation on all the pairs of operands stored in SIMD registers at a time. It adds a form of parallelism to software. It is a well-known technique for performance optimization. It makes the full use of the features provided by the hardware for parallelism. The aim of this work is to study the work done until now on vectorization and its related techniques for performance optimization.

Keywords: Vectorization, SIMD, Parallelism

I. INTRODUCTION

The Intel Xeon Phi coprocessor is based on the Intel Many Integrated Core (Intel MIC) architecture. It is innovative new processor architecture. It combines thread parallelism with long SIMD vector registers. Efficiently exploiting SIMD vector units is an important factor to achieve high performance of the application code running on Intel Xeon Phi coprocessors. The Intel MIC architecture consists of many power efficient, small cores. Each of these cores have a 512 bit long vector register. These vector registers are called SIMD (Single Instruction Multiple Data) unit. Because of this great width of SIMD units, we can make extensive use of SIMD vector operations. Hence such system is used for high performance computing workloads in many fields such as chemistry, biology, civil engineering, earthquake engineering and many more. Out of the top 500 supercomputers [8], 422 use Intel Xeon Phi Coprocessor. It is also used in research centers such as BARC (Bhabha Atomic Research Center). The Intel Xeon Phi Coprocessor has the key specifications as follows [7]:
- 60 cores,
- 4 hardware threads per core,
- 8 GB memory with 320GB/s bandwidth, 512 bit wide SIMD registers.
- 32kKB L1 cache and 512 KB L2 cache per core.
- Fused Multiply-Add support.

Many compute intensive applications trying to achieve high performance on Intel Xeon Phi coprocessors need to use the features provided by them to exploit the degree of parallelism[9]. Each Intel Xeon Phi is having 60 cores. So the application need to make itself scalable over the coprocessor for achieving performance gains. By using the 512-bit vector register, 16 single precision or 8 double precision floating point operations can be run at a single core per cycle. Also the FMA instructions can provide performance by running 32 floating point operations per cycle at single core. But wider SIMD vector registers cannot be used effectively simply by extending ISA of these processors such as Intel SSE and Intel AVX. This is where vectorization comes into picture. The goal of vectorization is to identify code regions that repeatedly apply given operations to consecutive data elements, and to rearrange the computation to benefit from SIMD extensions. Therefore, the vectorization technique is important for effectively utilizing the full length of SIMD vector units.

II. LITERATURE SURVEY

Utilizing full advantage of SIMD instructions in C programs is a tedious job and requires use of non-portable programming using intrinsics. In spite of the efforts took by developing auto-vectorization capabilities, this has to be done making all the efforts in vain. Whole Function Vectorization (WFV) [1] is a technique for extending the use of SIMD across entire functions. In this work, a vector-oriented programming framework is presented that facilitates WFV directly in C. This achieves performance which is more in comparison to that of performance gained by OpenCL and ISPC while maintaining Cs original syntax and semantics which allows C programmers to gain better performance for their applications by improving SIMD utilization, with-out stepping out of C. Efficiently exploiting SIMD vector units is one of the most important aspects in achieving high performance.

Efficiently exploiting SIMD vector units is one of the most important aspects in achieving high performance of the application code running. In the work presented in [2], several SIMD vectorization techniques such as less-than-full-vector loop vectorization, Intel MIC specific alignment optimization, and small matrix transpose/multiplication 2-D vectorization are implemented in the Intel C/C++ and FORTRAN compilers for Intel Xeon Phi coprocessors. A set of workloads from several application domains is
used conduct the performance benchmarking study of SIMD vectorization techniques. The performance results obtained had a performance gain of about 12.5 x.

Programmers write code using arrays and loops to perform complex computation. But loops are much slower than vector based optimization. Programmers first create and debug code originally written in arrays and loops construct. After that, they manually convert each loop into vector operation equivalent form. This job requires skilled programmer. It takes time, and introduces many errors. The work done in [3] presents an easy to use system. This system automatically detect for loops that can be vectorized. This allows programmer to select which loops to convert into a vector equivalent operation, debug the output and compare the performance so gained. All this is done in very interactive way with the intention of learning code vectorization. The functionality of Octave is extended with some additional operations which automatically detect loops for which it is possible to perform loop vectorization. All this work is performed with the help of web application. This application hides the Octave command line complexity and generates necessary octave command. Therefore, the user is able to use the tool to learn and use code vectorization tool with ease.

In most cases, vectorization modifies the program control flow. It modifies loop bounds of existing loops. It also adds new loops and new if-statements. From a worst case execution time estimation perspective as discussed in [4], guessing from the outside what the compiler has done is error-prone. Worst case execution time estimation form a worst case execution time bound from analysis of machine code by considering the flow information provided by the source code. This flow information helps to produce tight bound on execution time. But to maintain consistent flow information throughout is difficult. So the work done in [4] helps to maintain the consistent flow information when the optimization techniques and vectorization are applied. With the help of consistent flow information, the correct estimation of worst case execution time is possible. When the compiler vectorizes the code, the flow information is traced and updated accordingly. Then, the final flow information can be conveyed to worst case execution time analysis tools and used for the worst case analysis time calculation.

Data-parallel programming languages are an important component in parallel computing. Languages like CUDA or OpenCL are mostly used for general purpose parallelism. The implementations of those languages on CPUs rely wholly on multi-threading to implement parallelism. They ignore the additional intra-core parallelism provided by the SIMD instruction set of those processors (e.g. Intel’s SSE and the AVX or instruction sets). The work done in [5] discusses about several aspects of implementing data parallel languages on machines having SIMD instruction sets to exploit parallelism. The main contribution is a code transformation that is language and platform independent and that performs whole-function vectorization on low-level intermediate code.

Energy efficiency has become an important research area in the design of microprocessor. Parallelization, vectorization, specialization and heterogeneity are the key points of design to deal with the utilization wall. Heterogeneous architectures are enhanced with architectural optimizations, such as vectorization, for further increasing the energy efficiency of the processor thereby reducing the number of instructions which go through the pipeline and leveraging the usage of the memory hierarchy. The work done in [6] evaluates the energy efficiency of different processors from the Intel Core i7 and i5 family. It uses selected benchmarks from the PARSEC suite with variable core counts and vectorization techniques to measure the energy efficiency under the Thermal Design Power (TDP). Results thus obtained showed that it is better if software developers prioritize vectorization techniques over parallelization techniques whenever possible. It is much better in terms of energy efficiency. When using vectorization and parallelization together, scalability of the application reduces drastically. This may require different development strategies to maximize the utilization of resources for increasing energy efficiency.

The key to effectively utilize the custom computing systems is to make maximum use of their available parallelism. This job is difficult to achieve while keeping care of the hardware specific constraints. For the processors with fixed architecture, it is an easy job for vectorizing compiler to detect and exploit the parallelism. The work done in [10] presents a framework for producing optimized pipelined circuits from high-level programs. It combines the vectorization of inner loops with circuit pipelining to extract parallelism in a sequential program and to exploit this parallelism in hardware. The benchmarking done shows that the pipeline vectorization obtains more efficient circuits than sequential design techniques.

The work done in [11] introduces approaches that targets specific features of modern and upcoming SIMD Architectures. All presented approaches are recently realized in our source-to-source vectorizer Scout. Scout is configurable source-to-source transformation tool which automatically vectorizes the C source code. They presented approaches address issues encountered during the vectorization of production codes which are mainly from CFD domain. The performance measurements show that there is considerable speedup of codes auto-vectorized by Scout over traditionally vectorized codes. The programs can benefit from modern SIMD hardware more and more by enhancing auto-vectorization techniques with the discussed methods.

### III. WHY TO GO FOR VECTORIZATION

Full use of vector instructions is the ability to do 16 single precision or 8 double precision mathematical operations at once instead of doing one double or single precision mathematical operation. The performance boost that is obtained is substantial. Few years ago, Intel SSE, processors were able to offer only four single precision or two double precision operations per instruction. But with the launch of Intel’s AVX extension to the instruction set architecture now eight single and four double precision operations per instruction can be performed. Now that we have increased capacity for performing vectorization, a better performance gain could be obtained from Intel Xeon Phi coprocessors. To be vectorizable, loops need to satisfy some criteria which is described below:
1) The loop trip count must be known at entry to the loop at runtime.
2) The exit from the loop must not be data dependent.
3) There must be single entry and exit point.
4) SIMD instructions perform the same operation on data elements from many iterations of single loop. Therefore, each iteration should have similar control flow. There must not be any branches. Hence the presence of switch statement leaves the loop unvectorized.
5) Only the inner most loop of the nested loops is vectorized.
6) No function calls are allowed in the loop except the math functions or inline functions.

IV. TECHNIQUES FOR VECTORIZATION

These techniques involve packing multiple independent instructions together which can be run in parallel increasing resource utilization and performance:
1) Loop Unrolling: It involves executing multiple loop iterations at once on the device to increase utilization.
2) Two-way vectorization: It automatically extracts two-way short vector SIMD parallelism from a scalar code block by adequately combining scalar variables into SIMD variables and by joining the corresponding scalar instructions to one or more short vector SIMD instructions. Two-way vectorization transforms any pair of scalar instructions to one SIMD instruction yielding 100 percent SIMD utilization.
3) Data Alignment strategies: The Intel® Xeon Phi™ coprocessor is way a lot of sensitive to data alignment thus developing an data alignment strategy and optimization schemes is one amongst the key aspects for achieving optimum performance [12]. The compiler sharply performs data alignment optimizations using traditional techniques like alignment peeling. Alignment peeling implies the creation of a pre-loop that executes many iterations on unaligned data so as to achieve an aligned memory address. As a result, most of those iterations are executed using aligned SIMD operations.
4) Data dependence check: Data dependence check must be done in order to find out whether the two consecutive instructions have independent instructions to be executed [13].

V. OBSTACLE IN VECTORIZATION

There are some conditions due to which compiler prevents the vectorization because it would not be worthwhile in the case if following conditions are found true:
1) If the data objects are not in consecutive memory cells, then they must be loaded separately using multiple instructions. This causes overhead in gathering data in one single register having contiguous memory location. So the loops with non-contiguous memory access are not preferable for vectorization.
2) Each SIMD instruction operates on multiple data elements at once. Hence vectorization is possible only if changing the order of operations within a loop does not change the results of calculations.

VI. CONCLUSION

This paper has studied the significance of vectorization as one of the performance optimizing technique. It is synergistic. All cores should be kept busy to achieve peak performance. We need to know where vectorization should occur and verify that compiler is doing that. If a compiler fails to vectorize, then whether it is legitimate or not should also be checked. Applications can be vectorized to take advantage of SIMD instructions in Intel Xeon Phi and thus utilize the expanded vector width. There are no rules to achieve vectorization for gaining benefits from the overall program or application but the recommendations discussed in this paper increases the success rate of an application or program being vectorized.

REFERENCES

[11] Olaf Krzikalla1, Kim Feldhoff1, Ralph Miller-Pfefferkorn1, and Wolfgang E. Nagel1 “Auto-Vectorization Techniques for Modern SIMD Architectures”
