How Can We improve Energy Efficiency through User-directed Vectorization and Task-based Parallelization?

Helena Caminal, Diego Caballero, Juan M. Cebrián, Roger Ferrer, Marc Casas, Miquel Moretó,
Xavier Martorell and Mateo Valero
Barcelona Supercomputing Center

Abstract- *Heterogeneity, parallelization and vectorization are key techniques to improve the performance and energy efficiency of modern computing systems. However, programming and maintaining code for these architectures poses a huge challenge due to the ever-increasing architecture complexity. Task-based environments hide most of this complexity, improving scalability and usage of the available resources. In these environments, while there has been a lot of effort to ease parallelization and improve the usage of heterogeneous resources, vectorization has been considered a secondary objective. Furthermore, there has been a swift and unstoppable burst of vector architectures at all market segments, from embedded to HPC. Vectorization can no longer be ignored, but manual vectorization is tedious, error-prone, and not practical for the average programmer. This work evaluates the feasibility of user-directed vectorization in task-based applications. Our evaluation is based on the OmpSs programming model, extended to support user-directed vectorization for different SIMD architectures (i.e. SSE, AVX2, AVX512, etc). Results show that user-directed codes achieve manually-optimized code performance and energy efficiency with minimal code modifications, favoring portability across different SIMD architectures.*

Keywords SIMD, OmpSs, Performance, Vectorization, Energy Efficiency

I. INTRODUCTION

While transistor shrinking allows to include additional features on the die, the increasing power density prevents the simultaneous usage of all available resources. Instruction level parallelism (ILP) importance subsides, while data level parallelism (DLP) becomes a critical factor to improve the energy efficiency of microprocessors. Among other features, high performance computing (HPC). Each new generation includes more sophisticated, powerful and flexible instructions. The higher investment in SIMD resources per core makes extracting the full computational power of these vector units more important than ever.

From the programmers' point of view, SIMD units can be exploited in several ways, including: a) compiler auto-vectorization, b) low-level intrinsics or assembly code and c) programming models/languages with explicit SIMD support. Auto-vectorization in compilers has strong limitations in the analysis and code transformations phases that prevent an efficient extraction of SIMD parallelism in real applications. Low-level hardware-specific intrinsics enable developers to fine tune their applications by providing direct access to all of the SIMD features of the hardware. However, the use of intrinsics is time-consuming, tedious and error-prone even for advanced programmers. To facilitate the use of SIMD features, some programming models and languages have been extended with a new set of directives that allow programmers to guide the compiler in the vectorization process (e.g., OpenMP 4.0). This approach is high-level, orthogonal to the actual code and portable across different SIMD architectures.

In this abstract, we evaluate the efficiency of an implementation of a user-directed vectorization proposal using a task-based programming model. Our main contributions include:

- Development of a task-based version of a subset of benchmarks from the ParVec benchmark suite [2]. Due to space limitations we only show one of the six benchmarks we have ported.
- We present the code modifications necessary to generate a user-directed code version that achieves similar performance and energy results to those obtained with manual vectorization.
- We discuss our findings and propose improvements

zed versions and the user-

II. METHODOLOGY

In this document we evaluate three versions of the codes, including: a) two manually-vectorized implementations, one based on pthreads and one based on the OmpSs programming model [4] (labelled pthreads and OmpSs, respectively), and b) a user-directed vectorization (labelled U.D.). Both user-directed and OmpSs versions were developed for this document. The user-directed code is compiled using the Mercurium source-to-source infrastructure. Mercurium's vectorizer recognizes user annotations on the code to produce a SIMD version of the scalar code [1].

The evaluation platform is a dual-socket E5-2603v3 processor running at 1.60GHz, with a total of 12 cores, 30MB of L3 cache and 64GB of DDR3. We use PAPI to measure energy, L1D cache miss-rate and total instruction count. The reported energy numbers account for both sockets. The system runs CentOS 6.5 with Nanox 0.7.12a as runtime for the OmpSs codes.

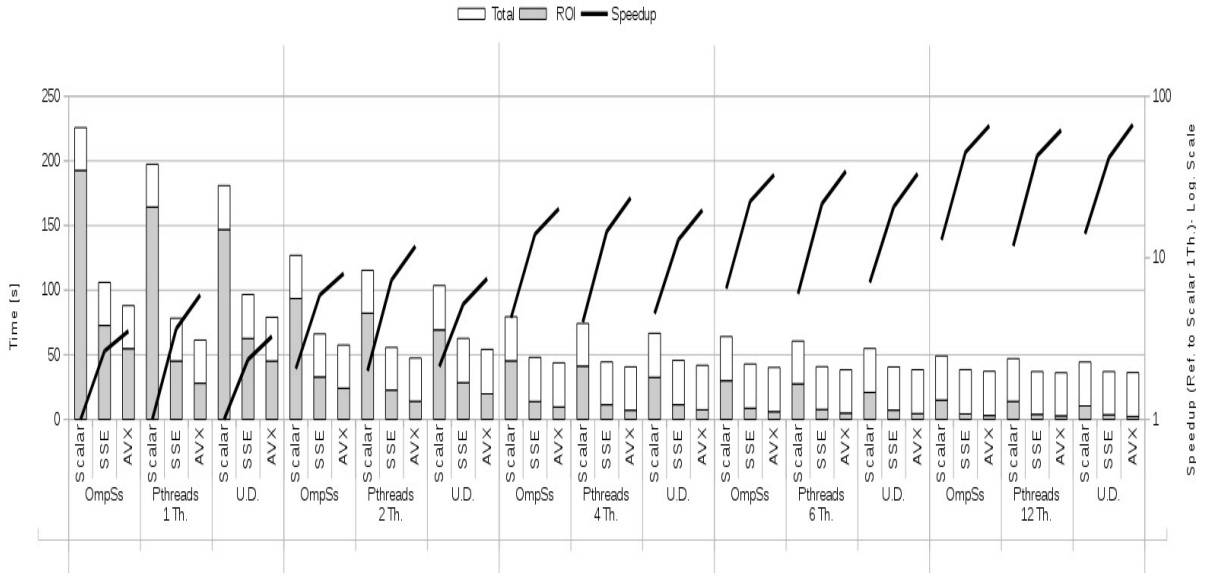

Fig. 1. Blackscholes runtime (Y axis) and speed-up (2nd Y axis

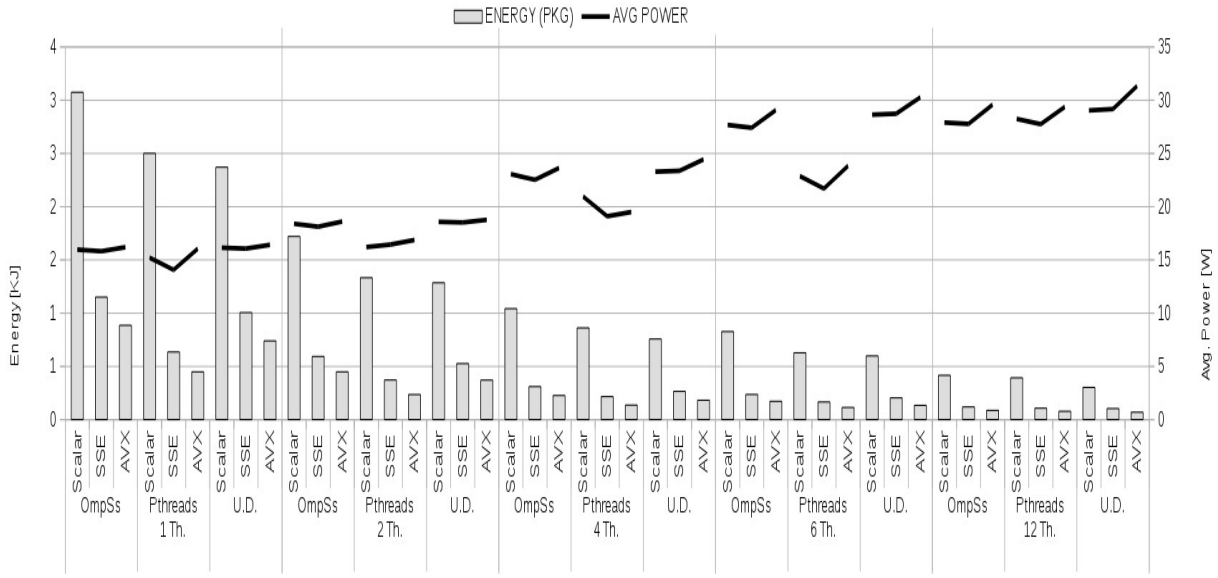

Fig. 2. Blackscholes energy consumption (Y axis) and power (2nd Y axis)

III. EVALUATION

This section shows performance and energy results for only one of the ParVec benchmarks [2] due to of space limitations. Execution times are shown in absolute numbers in order to compare performance between versions. In addition, speed-up is referenced to the scalar sequential
combination of each version to show scalability when varying thread count and vector length.

The blackscholes benchmark shows almost linear scalability with both thread count and vector length (Figure 1). This is mainly because of the high arithmetic intensity of the benchmark (computations per loaded data) and the low L1D cache miss-rate. Instruction count is also reduced linearly with vector length, meaning that we are vectorizing most of the application code.

Pthreads and OmpSs versions have the BlkSchlsEqEuroNoDiv and CNDF functions vectorized manually. In addition, some of the data structures have been aligned. Furthermore, the user-directed version only requires a single directive per function and loop to vectorize all 50 lines of code.

As shown in Fig. 2, power dissipation remains approximately constant in all SIMD versions. Intel platforms share both floating point registers and arithmetic units for scalar and SIMD instructions. While bit-toggling increases power

dissipation due to the extra vector length, the processor spends more time idle, waiting for data dependencies and memory operations, and thus dissipating similar average power independently of running scalar or SIMD code. Finally, it is worth mentioning that Nanos++ has an additional energy overhead when using one and two sockets. As threads spin while searching for work. In the Pthreads version, threads use blocking in the synchronization mechanisms.

IV. CONCLUSIONS

In this abstract, we present an evaluation in terms of performance and energy efficiency of user-directed SIMD implementations using a task-based programming model.

The application shows good performance scalability with vector length. The main reason for that is the reduction of executed instructions and memory accesses with respect to the scalar versions. Power dissipation remains constant when varying vector length. The blackscholes benchmark running with 12 threads can achieve energy improvements up to 35x. User-directed codes achieve similar performance and energy savings to those obtained with hand-vectorized code, while making the code portable between architectures and saving many lines of intrinsics code. As a result, we can confirm that vectorization together with parallelization are key techniques to improve energy efficiency.

REFERENCES

[1] D. L. Caballero de Gea, "PhD Thesis: SIMD@OpenMP: a programming model approach to leverage SIMD features." [Online].Available: http://www.tdx.cat/handle/10803/334171

[2] J. M. Cebrian, M. Jahre, and L. Natvig, "ParVec: Vectorizing the PARSEC Benchmark Suite," Computing, pp. 1077–1100, 2015.

[3] Programming Models, BSC, "The Mercurium C/C++ Source-to-source Compiler Website." [Online]. Available: http://pm.bsc.es/projects/mcxx

[4] A. Duran et al., "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architetcures," Parallel Processing Letters, vol. 21, pp.173–193, Mar. 2011.