The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

User-directed vs. Manual Vectorization: Performance and Energy Effects on Task-based Parallelized Applications

Helena Caminal · Diego Caballero · Juan M. Cebrián · Roger Ferrer · Marc Casas · Miquel Moretó · Xavier Martorell · Mateo Valero ·

Received: date / Accepted: date

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. 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 programmers. 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 Intel SIMD architectures (SSE, AVX2, IMCI and AVX-512). 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

1 Introduction

While transistor shrinking allows to include additional features and structures on the die, the increasing power density prevents the simultaneous usage of all available resources. Instruction level parallelism (ILP) importance subsides,

Helena Caminal, Juan M. Cebrián, Marc Casas, Miquel Moretó, Xavier Martorell, Mateo Valero
E-mail: first.last@bsc.es

Diego Caballero
E-mail: diego@ac.upc.edu

Roger Ferrer
E-mail: rofirrim@gmail.com

while data level parallelism (DLP) becomes a critical factor to improve the energy efficiency of microprocessors. Among other features, SIMD instructions have been gradually included in microprocessors for various market segments, from mobile (ARM NEON technology [1]) to high performance computing (Intel AVX-512 [2], ARM's Scalable Vector Extension [3] or PowerPC Altivec technology [4]). Each new generation includes more sophisticated, powerful and flexible instructions. This high investment in SIMD resources per core, specially in terms of area and power, 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 [5]. 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. Manual vectorization forces programmers to be knowledgeable about the offered SIMD instructions, and that becomes even more complicated with CISC ISAs. To facilitate the use of SIMD features, some programming models and languages have been extended to allow programmers to guide the compiler in the vectorization process. For example, OpenMP 4.5 [6] offers a set of directives to describe vectorizable regions. This approach is high-level, orthogonal to the actual code and portable across different SIMD architectures.

The OpenMP 4.5 standard supports tasking and data dependencies. Parallelism is described by a directed acyclic graph where each node is a task and the edges between nodes represent dependencies, which are explicitly annotated by the programmer. Such annotations also provide the opportunity for the runtime system to automatically offload tasks to accelerators like GPU's or Intel Xeon Phi co-processors. The runtime system is empowered to take care of data movements without the need of specific programming intervention besides annotating each task input and output dependencies. Also, the runtime system may deploy some optimizations like data prefetching or overlapping of computation and communication. It also enables the possibility to exploit data locality in distributed-cache architectures, by allocating computational resources near the cache partition where data resides.

In this article, we evaluate the efficiency of user-directed vectorization using OmpSs [7], developed at the Barcelona Supercomputing Center. OmpSs is a data-flow programming model, similar to OpenMP, that eases application porting to the heterogeneous architectures. Nanos++ [7] is used as runtime system for the OmpSs programming model. OmpSs offers advanced features like socket-aware scheduling for NUMA architectures or pragma annotations to handle multiple dependence scenarios Both models (OpenMP and OmpSs) have virtually the same syntax, thus porting OpenMP code to OmpSs and vice versa is straightforward.

Our main contributions include:

- Development of a task-based version of a subset of benchmarks from the ParVec benchmark suite [8]. As discussed by ParVec authors, benchmarks can be classified in scalable (S), resource limited (RL) and code/input limited (CI). We chose representative benchmarks that cover this classification: Blackscholes (S), Canneal, Streamcluster (RL) and Swaptions (CI).
- 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 proposed improvements for both the manually vectorized versions and the user-directed vectorization module in the Mercurium [9] source-to-source compiler.

This article is organized as follows. Section 2 introduces our evaluation methodology. Section 3 shows our main experimental results and discussion. Section 4 presents a brief summary of the related work on SIMD benchmarking and programming models. Finally, Section 5 shows our concluding remarks and future work.

2 Methodology

In this paper we evaluate three versions of codes, including: a) two manually-vectorized implementations, one parallelized with the pthreads programming model [8] and one parallelized with the OmpSs programming model [7] (labeled **pthreads** and **OmpSs**, respectively), and b) a user-directed vectorization which is also based on OmpSs (labeled **U.D.**) We initially tested automatic vectorization on the original scalar code but it resulted in no performance or energy improvements. Both user-directed and OmpSs versions were developed for this paper. Within the three versions, we have targeted the same loops and functions for vectorization, being the performance and energy consumption of the three versions comparable.

The pthreads codes have been compiled with ICC 14.0 and both the OmpSs and the U.D. codes are compiled with the Mercurium compiler [9]. Mercurium is a research source-to-source compiler with support for C, C++, and FORTRAN programming languages, and OpenMP [6], OmpSs [7] and StarSs [10] programming models, among others. We have extended the Mercurium source-to-source infrastructure to support the directives used in the U.D. codes [11]. Mercurium's vectorizer recognizes user annotations on the code to produce a SIMD version of the scalar code. Binaries are then built and linked using the Intel Compiler C/C++ as a back-end. We use `-no-vec` flag to isolate our results from the automatic vectorization performed by the Intel compiler. Further details on the building infrastructure can be found in Table 1. For each benchmark, we only take measurements from the Region of Interest (ROI) to ignore the initialization and finalization parts of the applications.

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.

	pthreads	*OmpSs, U.D.*
Front-end compiler	Intel Compiler C/C++ 14.0.1	Mercurium
Back-end compiler		Intel Compiler C/C++ 14.0.1
Flags C/C++ codes [C++ codes]	-03, -no-vec, -funroll-loops -qopt-prefetch, [-fpermissive, -fno-exceptions]	
Mathematical libraries	Short VectorMath Library(SVML)	

Table 1 Building infrastructure and configuration for *pthreads*, *OmpSs* and *U.D.* codes.

We use PAPI [12] to measure energy, L1D/L2/L3 cache miss-rate and total instruction count. The E5-2603v3 only provides energy information for the whole socket, since the power plane 0 is disabled (the one that offers energy results for the cores). The reported energy numbers account for both sockets. The system runs CentOS 6.5 with Nanox 0.7.12a as runtime for OmpSs.

We have tested three different input sizes for the benchmarks: *native*, *simlarge* and *simsmall*. Overall, L1, L2 and L3 cache measurements are affected by the input size, emphasizing the differences between the OmpSs/U.D. versions versus the pthreads version. For that reason, we encourage the research community to use the largest inputs possible even when they are using simulation tools. Results will be clearer and more significant for the different programming models. In terms of vectorization, bigger input sizes usually favor the use of longer vectors leading to a better performance and energy improvements. Nevertheless, this depends on the application's algorithm.

Manual Vectorization: The manually vectorized (pthreads and OmpSs) codes make use of a wrapper library [13] that provides generic vector intrinsics. These intrinsics are translated to architecture-specific intrinsics at compile time. Vector instructions that are not supported in the target ISA are emulated to have the same functionality. For further details on ParVec benchmark specifics and the manual vectorization process refer to [8] for further details. Figure 2 (bottom) shows the manually vectorized version of the **dist** function (top) included in the streamcluster benchmark. Note that the transformation is based on a direct translation from the scalar operations (-, *, +) to their equivalent vector intrinsic (*_MM_SUB*, *_MM_MUL*, *_MM_ADD*). The library will then translate those calls to ISA-specific intrinsics (e.g., _MM_ADD to _mm_add_ps for floats in SSE, or _mm256_add_ps for floats using AVX). This increments portability across different architectures and abstracts the low-level details to the programmer. In addition, we do a vector load (*_MM_LOADU*) and we increment the iteration count by a generic *SIMD_WIDTH*.

User-Directed Vectorization: The vectorization infrastructure implemented in Mercurium is divided in two main phases: *Vectorizer* and *Vector Lowering*. Vectorizer is in charge of transforming the scalar input code into a generic vector representation, within the compiler middle-end stage. Later, the vector

```
float dist(Point p1, Point p2, int dim) {
  int i;
  float result=0.0f;
  for (i=0;i<dim;i++) {
    result += (p1.coord[i] - p2.coord[i]) * (p1.coord[i] - p2.coord[i]);
  }
  return(result);
}
```

```
float dist(Point p1, Point p2, int dim) {
  int i;
  _MM_TYPE result, _aux, _diff, _coord1, _coord2;
  result = _MM_SETZERO();
  for (i=0;i<dim;i=i+SIMD_WIDTH) {
    _coord1 = _MM_LOADU(&(p1.coord[i]));
    _coord2 = _MM_LOADU(&(p2.coord[i]));
    _diff = _MM_SUB(_coord1, _coord2);
    _aux = _MM_MUL(_diff, _diff);
    result = _MM_ADD(result, _aux);
  }
return((float)_MM_REDUCE_ADD(result));
}
```

Fig. 1 Manual vectorization example over C code (top) using the wrapper library.

```
#pragma omp simd[clause[clause] ...] new-line
  for-loop -- function-decl -- function-def
```

Fig. 2 C/C++ syntax of the standalone *simd* construct.

lowering phase generates architecture specific SIMD intrinsics. The vectorization algorithm is based on the traditional strip-mining/unroll-and-jam loop vectorization approach [14,15]. This algorithm vectorizes two kinds of code structures: loops and functions. The simplest construct to describe SIMD parallelism is the `pragma omp simd` directive, placed on top of one of these code structures (Figure 2). This directive is used to instruct de compiler to vectorize the code, relaxing some restrictions that otherwise would prevent its vectorization. For that purpose, the compiler will assume that the vectorization is safe and profitable without running any legality and cost model analyses. OmpSs provides optional clauses to offer further information to the compiler about the target code, such as the `aligned` clause, the `suitable` clause and the `vectorlength` clause, among others. More detail will be given on the clauses used for each benchmark case. Please, refer to Caballero de Gea's work [11] for further details on Mercurium's vectorizer. Figure 3 shows a use case of the compiler directive on the **dist** function of the streamcluster benchmark. The addition of the optional clause `reduction(+:result)` to the standalone directive annotating the loop statement is enough to automatically vectorize the code. The reduction clause follows the same style as the parallel constructs used for threads. It generates a scalar result to be stored in the `result` variable using the specified reduction operation (`+`) on to the scalar values of each lane. Redundant instructions are combined by the Backend.

```
float dist(Point p1, Point p2, int dim) {
  int i;
  float result=0.0f;
  #pragma omp simd reduction(+:result)
  for (i=0;i<dim;i++) {
    result += (p1.coord[i] - p2.coord[i]) * (p1.coord[i] - p2.coord[i]);
  }
  return(result);
}
```

```
float dist(::Point p1, ::Point p2, int dim) {
  int i;
  float result(0.00000000000000000000000e+00f);
  {
    __m256 __vred_result(_mm256_set1_ps(0));
    for (i = 0; i <= -8 + dim; i = 8 + i)
      {
        __m256 _v3atmp0;
        _v3atmp0 = _mm256_loadu_ps(&p1.coord[i]);
        __m256 _v3atmp1;
        _v3atmp1 = _mm256_loadu_ps(&p2.coord[i]);
        __m256 _v3atmp2;
        _v3atmp2 = _mm256_loadu_ps(&p1.coord[i]);
        __m256 _v3atmp3;
        _v3atmp3 = _mm256_loadu_ps(&p2.coord[i]);
        __m256 _v3atmp4;
        _v3atmp4 = _mm256_sub_ps(_v3atmp0, _v3atmp1);
        __m256 _v3atmp5;
        _v3atmp5 = _mm256_sub_ps(_v3atmp2, _v3atmp3);
        __m256 _v3atmp6;
        _v3atmp6 = _mm256_mul_ps(_v3atmp4, _v3atmp5);
        __vred_result = _mm256_add_ps(__vred_result, _v3atmp6);
      }
#pragma loop_count min(0) max(7)
    for (; i <= -1 + dim; i = 1 + i)
      {
        result = result + (p1.coord[i] - p2.coord[i]) *
                          (p1.coord[i] - p2.coord[i]);
      }
    result += ({
      ::__m256 __rtmp256
      ::__m128 __rtmp128
      __rtmp256 = _mm256_hadd_ps(__vred_result, __vred_result);
      __rtmp128 = _mm_add_ps(_mm256_castps256_ps128(__rtmp256),
                             _mm256_extractf128_ps(__rtmp256, 1));
      __rtmp128 = _mm_hadd_ps(__rtmp128, __rtmp128);
      _mm_cvtss_f32(__rtmp128);
    }) ;
  }
  return result;
}
```

Fig. 3 Mercurium source-to-source input (top) and output (bottom). Redundant instructions are combined by the Backend.

Mercurium's vectorization functionalities are still a work in progress, and as such, some of the situations require some code preparation to be done by the programmer. First, aligned accesses yield significantly better performance. Mercurium can generate unaligned loads, but on some platforms, like Intel's Xeon Phi, performance can be compromised [16]. Second, Mercurium supports vectorization of "if-then-else" statements using predication. Current implementations rely on mask registers to decide which lanes of the vector

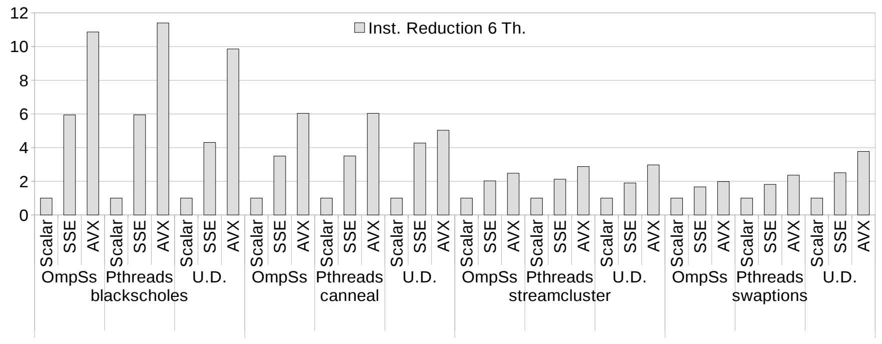

Fig. 4 All benchmarks instruction count reduction. It does not change with thread count, thus, only the configuration with a 6-thread configuration is shown.

register perform the specific operation. This feature is supported by new architectures, such as AVX2 (predicated loads/stores), AVX-512 and SVE, but not in SSE or NEON. As such, Mercurium will not be able to directly produce code for these architectures. However, Mercurium able to vectorize ternary operators using blend operations. Therefore, "if-then-else" statements need to be transformed into simple ternary operators to be vectorized for old architectures. Third, similar to manual vectorization, it is recommended to transform data structures from array-of-structures (AoS) into structure-of-arrays (SoA) Although there is some ongoing work to automatize this process [17,18], we reused the transformations that were already applied by the ParVec's authors.

3 Evaluation

This section shows performance and energy measurements for a subset of the ParVec benchmarks [8]: blackscholes, canneal, streamcluster and swaptions. Each application has been executed and analyzed with three programming models (pthreads, OmpSs and U.D.) and three instruction sets (scalar, SSE, AVX). Speed-up and energy reduction are referenced to the scalar sequential combination of each version to show scalability when varying thread count and vector length. Figure 4 shows the ratio of instruction count reduction normalized to the baseline scalar code for all applications when running on 6 threads. Similar results are achieved for other thread counts. Scalability with SIMD register size is similar for all three implementations, meaning that user-directed vectorization can achieve similar results to manual vectorization.

It is important to note that Intel platforms share both floating point registers and arithmetic units for scalar and SIMD instructions. While longer registers burn more power, if we reduce enough the execution time we can reduce the overall energy comsumption. Additonally, instruction count is reduced with SSE and AVX compared to a scalar version, reducing the front-end pressure of the pipeline. The core also spends more time idle, waiting for memory operations and data dependencies to be resolved. As a result, power dissipation remains approximately constant in all SIMD and scalar versions, as we show in the following sections. Therefore, any performance improvement from vectorization will come "for free" in terms of power, leading to substantial energy savings. In figures 5, 7, 9 and 11 we show speedup with respect to the scalar-1 thread configuration of each code version (OmpSs, pthreads, U.D.), average power for the whole execution, and energy reduction defined as the energy consumed through the whole execution normalized to the scalar-1 thread configuration of each code version.

3.1 Blackscholes

The hot regions to be vectorized are functions `BlkSchlsEq- EuroNoDiv` and `CNDF`. For the **pthreads** and **OmpSs** codes, these functions account for roughly 50 lines of intrinsics per instruction set (SSE, NEON, AVX). All this manually vectorized code can be avoided by using the SIMD directives. In order to instruct the compiler to vectorize the targeted functions, we annotated them with no additional clause above the definition of the function `CNDF` and the internal loop in function `BlkSchlsEqEuroNoDiv`. C-standard math library calls are replaced with Intel's Short Vector Math Library calls in all codes, to further improve performance. As in all of the studied codes, most of the data structures have been aligned to vector length boundaries. The **U.D.** code also needs a conversion of "if-then-else" statement to a ternary operator.

The blackscholes benchmark shows almost linear scalability with both thread count and vector length (Figure 5). This is mainly because of the high arithmetic intensity of the benchmark (computations per amount of loads) and the low L1D cache miss-ratio (Figure 6). Using vector instructions reduces time in the Region of Interest (ROI) by a range of 2.7x to 4.3x for SSE instructions and 3.5x to 6.9x for AVX. We observe that the manually vectorized and the user directed versions scale in a similar manner. Instruction reduction (Figure 4) is above the theoretical optimal factor (4x for SSE and 8x for AVX). The reason is that operations such as the *logarithm* and the *exponential* are implemented by the SVML library and are not a direct translation from the scalar instructions in the C library.

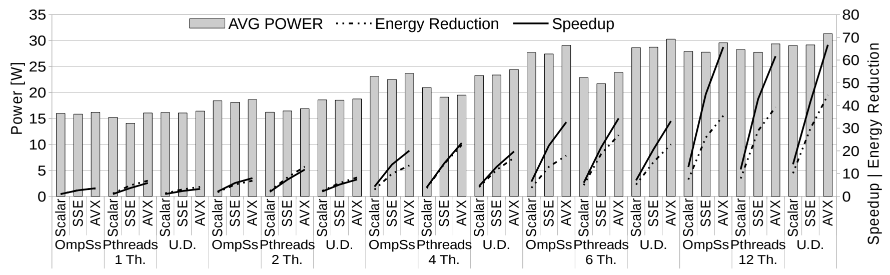

Fig. 5 Blackscholes power dissipation (left Y axis) and speedup/energy reduction factor (right Y axis) for different core count and SIMD instruction sets.

Energy is reduced close to a factor of 3x for OmpSs and 5x (per thread) for pthreads (as shown in Figure 5), most likely due to the overhead of the Nanos++ runtime when distributing work among threads. This leads to an energy reduction of 40x when running on 12 threads. Finally, it is worth mentioning that Nanos++ has an additional energy overhead when using two sockets. This is due to the threads spinning while searching for work.

3.2 Canneal

We are using a clustered version of this benchmark (as it was in the ParVec benchmark suite [8]) to increase computational density, thus improve the vectorized performance. For this application we performed an array-of-structure (AoS) to structure-of-array (SoA) data conversion for all three versions.

For the OmpSs and U.D. (task-based) codes, scalability is linear with thread count and almost linear for pthreads (Figure 7). Performance also increases linearly with the vector length for manually vectorized versions with up to 1.56x (2.00x), but is far from the optimal 4x (8x) for SSE (AVX), respectively. The use of SIMD instructions increases the miss-rate of the L1D cache (Figure 8) by a factor of 4x for SSE instructions and by a factor of 6 for AVX instructions, becoming a significant bottleneck for scalability as we increase the vector length. The AVX performance on the U.D. version does not achieve the same performance and instruction count reduction as the pthreads and OmpSs relatives. The handcrafted vectorized versions (pthreads and OmpSs) are more sophisticated than the U.D., their vectorization strategy is personalized for the targeted vector extensions (either SSE or AVX). Instead, the U.D. code generates the exact same vector code except for the register size. The 16x performance gain translates to energy improvements by a factor of 10x when using AVX for 12 threads. This is 4x more than when only relying on threading (Figure 7).

For the U.D. code, two loops are vectorized inside function *swap_cost-_block_omp_simd* (which is the SIMD version of *swap_cost_block* function). Additionally to the standalone construct, we also add the optional clause `aligned( loc_cluster_x, loc_cluster_y)`. This clause is used to inform the

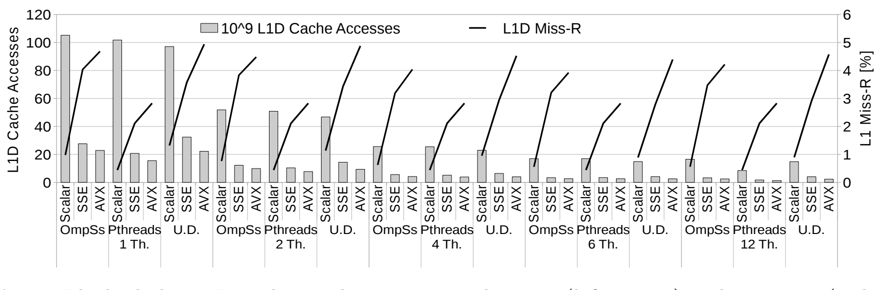

Fig. 6 Blackscholes L1D cache total accesses, total misses (left Y axis) and miss rate (right Y axis).

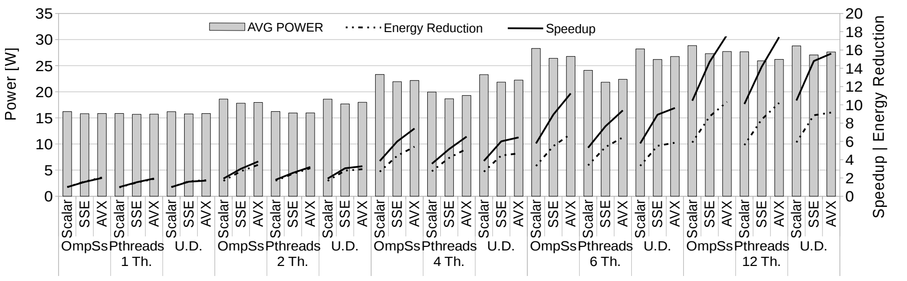

Fig. 7 Canneal power dissipation (left Y axis) and speedup/energy reduction factor (right Y axis) for different core count and SIMD instruction sets.

compiler that `loc_cluster_x` and `loc_cluster_y` are aligned to the vector length boundary of the architecture. We aligned these structures to the respective vector length used in each case (16 bytes for SSE or NEON and 32 bytes for AVX) when we transformed the memory layout of the data structure from AoS to SoA. Without the clause, the compiler would assume that the pointers are not aligned to any specific boundary. As a consequence, it would generate unaligned vector memory accesses, that may translate into poor performance.

3.3 Streamcluster

This application has almost linear scalability up to 6 threads (Figure 9). However, it does not scale linearly when using two sockets, due to the significant number of synchronization barriers. This can be explained by the slowdown of the interconnection network compared to shared memory. Function **dist** was vectorized as it accounts for most of the execution time, which has a small set of arithmetic operations. As a consequence, scalability depends on the memory subsystem accompaniment. Performance increases by a factor of 1.8x and 2.1x when using SSE and AVX instructions, respectively, with regard to the scalar version. This means that we are unable to fully benefit from AVX due to the memory subsystem.

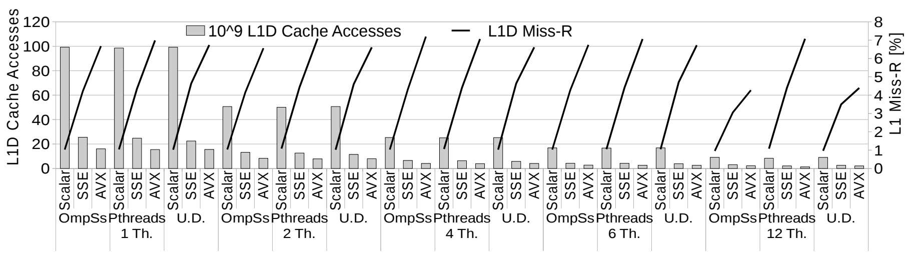

Fig. 8 Canneal L1D cache total accesses, total misses (left Y axis) and miss rate (right Y axis).

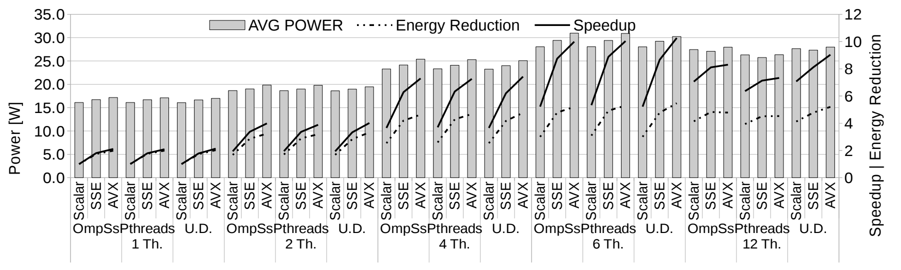

Fig. 9 Streamcluster power dissipation (left Y axis) and speedup/energy reduction factor (right Y axis) for different core count and SIMD instruction sets.

While energy scales linearly with thread count until full socket use, performance scalability issues with vector length translates into sub-linear energy improvements (5x for AVX as compared to 3x without SIMD in the case of 6 threads). Consumption, speedup and energy reduction when using two sockets is explained by the time spent in inter-socket barrier synchronization.

As shown in Figure 3 (top), the function **dist** is annotated to vectorize the code. `reduction(+:result)` clause is added to instruct the compiler to perform a horizontal reduction on variable `result` using `+` operator. This annotation is enough to vectorize the loop and generate similar code to the pthreads and OmpSs codes.

3.4 Swaptions

Swaptions benchmark shows linear scalability with thread/task count, but limited scalability with vector length (Figure 11). In fact, instruction count is reduced by a factor of 1.75x to 2.5x for SSE and AVX, respectively (down from a potential 4x for SSE and 8x for AVX). In addition, swaptions performance is also limited by multiple data dependencies and high L1D cache miss-rate (1.5% for scalar, 2.5% for AVX), as shown in Figure 12. Energy follows performance closely (Figure 11), again experiencing a moderate increase in average power

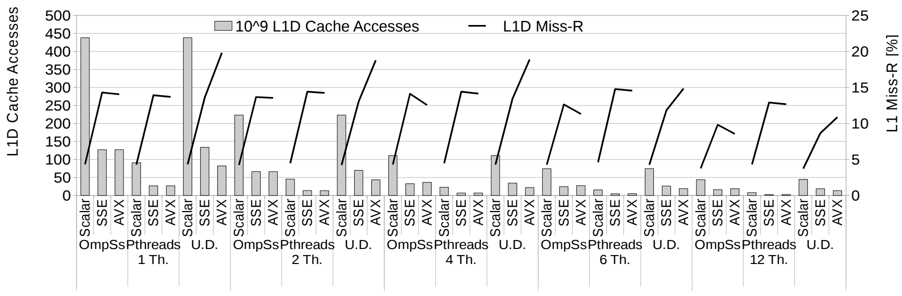

Fig. 10 Streamcluster L1D cache total accesses, total misses (left Y axis) and miss rate (right Y axis).

due to the Nanos++ runtime system. The increase on vector scalability due to predication of the code yields moderate energy benefits for SSE and AVX on the U.D. code. AVX instructions manage to almost double the energy efficiency of the application.

This application has several complex "if-then-else" statements that Mercurium's vectorizer is unable to handle efficiently, so we decided to manually predicate the code on the U.D. version. In this way, arithmetic operations are executed unconditionally and conditions are only used on memory operations. This causes a significant increment of the execution time of the scalar version of the U.D. codes, but makes the SSE and AVX versions scale better with vector length (as compared with the manual vectorization with "if-then-else" statements). We plan to rewrite the manual vectorization (in pthreads and OmpSs codes) in our future work using the same strategy. The U.D. code saves around 20 lines of instrinsic-based code plus a big vector initialization phase that is specific for the target instruction sets (SSE and AVX, around 40 lines of code), making it suitable for any other architecture. In addition to the manual predication of the U.D. codes, we used the standalone construct in all of the loops inside *HJM_SimPath_Forward_Blocking* function. The construct was accompanied with the `aligned(ppdHJMPath[:])` and `suitable(BLOCKSIZE)` clauses. The clause `suitable`: it is used to inform the compiler that the variable `BLOCKSIZE` will have a value at runtime that will be multiple of the vector length of the architecture. The use of `suitable` together with `aligned` clauses

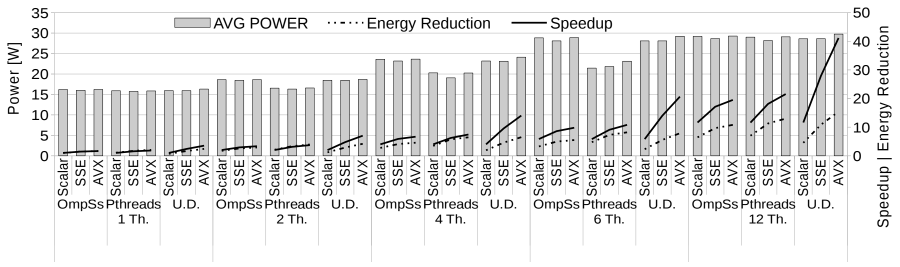

Fig. 11 Swaptions power dissipation (left Y axis) and speedup/energy reduction factor (right Y axis) for different core count and SIMD instruction sets.

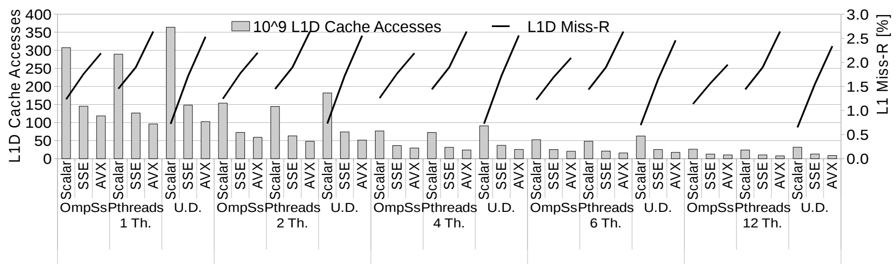

Fig. 12 Swaptions L1D cache total accesses, total misses (left Y axis) and miss rate (right Y axis).

improves the vectorization of certain loops with multi-dimensional arrays. In this case, the expression inside the loop `ppdHJMPath[0][BLOCKSIZE*j + b]` is computed as `ppdHJMPath + BLOCKSIZE*j + b`, which is aligned to the vector length boundary.

3.5 Effects of Hardware Prefetching

Hardware prefetching becomes a critical component on SIMD architectures, since SIMD applications significantly increase the pressure on the memory subsystem. We cannot provide results for the L1D cache since we cannot interact with the prefetcher at this level, and did not find an alternative way of disabling the prefetching mechanism for that L1D cache level. As for the results for L2 cache level (Figures 13 and 14), we see similar results for the OmpSs and the U.D. versions and different than the one for pthreads. Both for OmpSs and U.D. versions, the number of accesses is significantly less and at the same time the absolute number of misses is much higher without prefetching. These two facts are not surprising and just confirm that the prefetcher for that cache level works as it is expected. On the other hand, the pthreads version has significantly less accesses to the L2 cache compared to the other two versions (specially for the case of streamcluster). This fact can be explained by the way the Nanos++ runtime works, creating the dependence task graph and incrementing the number of memory accesses. Moreover, when we disable the prefetching mechanism we see that most of the accesses are misses. Results for the L3 cache level are similar to the ones for L2 and can be explained by the same factors.

For blackscholes, prefetching has a significant impact on performance (and consequently on energy efficiency). The speedup per thread of the native inputs on the pthreads version compared to the scalar OmpSs version goes up from 1.2054x for scalar, 3.7543x for SSE and 5.7072x for AVX to 1.2398x, 4.5159 7.3109x respectively. Canneal barely benefits from the hardware prefetching, going from 0.9882x for scalar, 1.5099x for SSE and 1.9251x for AVX to 0.9808x, 1.5292x and 1.9369x respectively. The same happens with swaptions, going from 1.0810x, 1.6916x and 2.0119x to 1.0882x, 1.6999x and 2.0237x respectively. Finally, results for streamcluster show the opposite trend. For this benchmark, hardware prefetching is actually harming performance, going from 1.0048x for scalar, 2.2819x for SSE and 2.8350x for AVX to 1.0117x, 1.8097x and 2.1074x respectively. This behavior is exacerbated for smaller inputs, performing almost twice as fast without prefetching for *simlarge*. This results confirm the urgent need to apply manual prefetching for this benchmark.

Globally, the binaries executed with prefetching mechanism obtain higher energy reduction as the memory accesses result in a higher number of hits. The power consumption is approximately constant while varying the register length. With the instruction count reduction previously mentioned and the power consumption remaining constant, we clearly obtained an energy reduction proportional to the speedup.

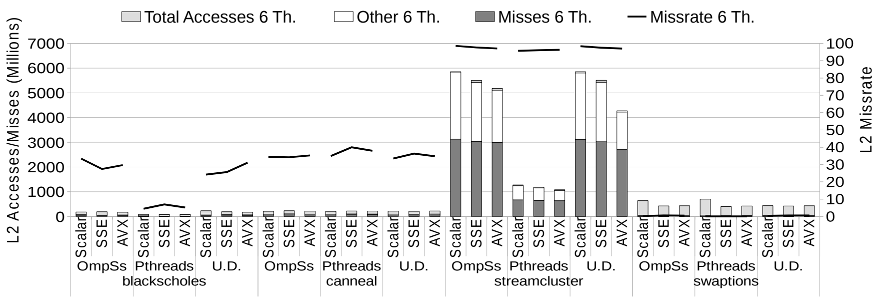

Fig. 13 All benchmarks L2 total access count, miss count, other type of access count and miss ratio with 6-thread configuration with prefetching mechanisms enabled.

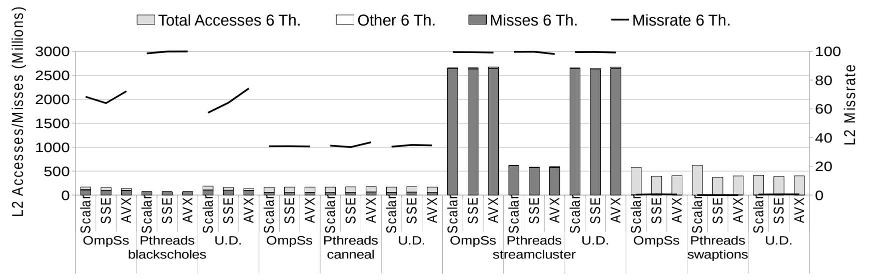

Fig. 14 All benchmarks L2 total access count, miss count, other type of access count and miss ratio with 6-thread configuration with prefetching mechanisms disabled.

4 Related Work

New programming models and languages with support for vector data and vector operations have emerged in the last years. Among them, the most popular are OpenCL and CUDA, which define vector data types to describe vector operations that will be automatically vectorized by the compiler. In addition, ISPC [19] is a SPMD programming model that allows programmers to natively write applications with SIMD parallelism in mind. Chorus [20] extends C with the map and fold functions commonly used in Functional Programming Chorus, but they target them to vector operations. Although these approaches could yield good performance and more portability, sometimes not only a deeper code rewriting might be needed, but also a full redesign of the applications.

On the other hand, user-directed vectorization exploits the power of SIMD instructions with minimal code modifications. There are several proposals based on compiler directives that allow programmers to propose loops that should be vectorized by the compiler. SIMD extensions defined in OpenMP 4.5 and OpenACC 2.0 are two examples of them. In this way, programmers can guide the vectorization of their code and they can benchmark different vector versions by means of introducing simple annotations in their applications. These approaches require little effort by the programmer whereas they still provide very good performance across platforms. Our work pursues the same goals, but we define SIMD extensions in the context of the OpenMP programming model. OmpSs extensions also allow annotating a loop as safely vectorizable but we go a step further. We define the interaction between SIMD parallelism and fork-join parallelism in an integrated parallel approach. Nowadays, OpenMP includes SIMD extensions to express SIMD parallelism. We collaborated in the definition of the extensions that are now part of the 4.0 standard, in the same way that OmpSs was the building block for the task support in OpenMP.

Regarding SIMD benchmarking, Molka et al. [21] discuss weaknesses of the Green500 list with respect to ranking HPC system energy efficiency. They introduce their own benchmark using a parallel workload generator and SIMD support to stress main components in a HPC system, but do not consider a task-based scenario. Kim *et al.* [22] show how blocking, vectorization and minor algorithmic changes can speed up applications close to the best known tuned version. Our work evaluates if task-based benchmarks with user-directed vectorization can behave in the same way. The RODINIA [23] and ALPBench [24] benchmark suites also offer limited SIMD support, but not in the same scenario that we explore. Cebrian et al. [8] extended the Parsec benchmark suite to add SIMD support using manual vectorization. We will use these benchmarks as our starting point to build user-directed/task-based versions and compare them in terms of performance and energy. The goal is to produce similar quality code without the need of low-level programming.

5 Conclusions and Future Work

The main contribution of this paper is to compare different vectorization strategies and scenarios that would ease the vectorization process of end user applications and libraries. Our goal is to validate if user-directed vectorization, such as the one in OmpSs (or equivalent) can produce similar quality code to the one obtained by manual vectorization introducing annotations in the code. We show results for four ParVec benchmarks parallelized with both OmpSs (which we have developed for this publication) and pthreads programming models. Mercurium source-to-source infrastructure is used to obtain a SIMD version of the application based on annotated scalar code. We use highly optimized manually vectorized versions as a reference to leverage the results of the user-directed codes. We describe the necessary code modifications and compiler directives used to obtain the user-directed codes that achieve a similar performance and energy efficiency to the manually vectorized codes.

The evaluation shows good energy scalability with vector length, specially for blackscholes and canneal benchmarks. The main reason for that is the reduction of executed instructions and memory accesses with respect to the scalar versions. The combination of the task-based benchmarks plus the user-directed vectorization provides similar reduction on run-time while maintaining the power consumption approximately the same. This is explained by the fact that Intel architectures share register and arithmetic units for scalar and SIMD instructions. Power dissipation increases with vector length due to additional bit-toggling, but the processor spends more idle waiting for the memory subsystem. As a consequence, average power remains similar while energy is reduced superlinearly with runtime.

We have showed that scalable applications running with 12 threads can achieve energy improvements up to 40x (blackscholes) while other applications can still benefit from up to 5x (streamcluster). As a result, we can confirm that vectorization together with parallelization are key techniques to improve energy efficiency.

As vector support is already existent in most commodity processors, we can reduce energy only by learning how to vectorize our applications in a comfortable manner. Both parallelization and vectorization can be achieved without the need of low level programming, by using task and vector annotations (pragmas) on the code. In addition, user-directed vectorization keeps the abstraction layer with the underlaying architecture, making the code portable between architectures and saving many lines of intrinsics code, though it may require reorganizing or modifying the code.

As future work, we aim to improve on energy awareness for runtime systems running on multisocket machines. Our future line of work will be to extend the evaluation of user-directed performance on other applications while we extend Mercurium with additional features. At last, we would like to replace Intel's Short Vector Math Library (SVML) with a more generic one in order to support other ISA's (e.g. ARM Scalable Vector Extension).

References

1. A. Limited, "ARM NEON support in the ARM compiler," 2007.
2. N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "White Paper: Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency," 2008.
3. N. Stevens, "The scalable vector extension (SVE) for the ARMv8-A architecture," in *Hot Chips'16*, 2016.
4. S. Fuller, "Motorolas AltiVec Technology," 1998.
5. S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 372–382, IEEE Computer Society, 2011.
6. OpenMP ARB, "OpenMP 4.0/4.5 Specifications."
7. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architetcures," *Parallel Processing Letters*, vol. 21, pp. 173–193, Mar. 2011.
8. J. M. Cebrian, M. Jahre, and L. Natvig, "ParVec: Vectorizing the PARSEC Benchmark Suite," *Computing*, pp. 1077–1100, 2015.
9. Programming Models, Barcelona Supercomputing Center, "The Mercurium C/C++ Source-to-source Compiler Website."
10. E. Ayguad, R. M. Badia, P. Bellens, J. Bueno-Hedo, A. Duran, Y. Etsion, M. Farreras, R. Ferrer, J. Labarta, V. Marjanovic, L. Martinell, X. Martorell, J. M. Prez, J. Planas, A. Ramirez, X. Teruel, I. Tsalouchidou, and M. Valero, *Hybrid/Heterogeneous Programming with OmpSs and its Software/Hardware Implications*. Programming Multi-Core and Many-Core Computing Systems. John Wiley and Sons, Inc., Wiley Series on Parallel and Distributed Computing edition, 2012.

11. D. L. Caballero de Gea, “PhD Thesis: SIMD@OpenMP : a programming model approach to leverage SIMD features.”
12. P. J. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: A Portable Interface to Hardware Performance Counters,” in *Proc. of the Department of Defense HPCMP Users Group Conference*, 1999.
13. L. N. Juan M. Cebrian, Magnus Jahre, “Optimized hardware for suboptimal software: The case for simd-aware benchmarks,” in *Proceedings of 2014 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2014.
14. K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.
15. A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, “Automatic intra-register vectorization for the intel architecture,” *Int. J. Parallel Program.*, vol. 30, pp. 65–98, Apr. 2002.
16. Intel Corporation, “Intel Knights Corner.”
17. N. Sharma, P. R. Panda, F. Catthoor, P. Raghavan, and T. V. Aa, “Array interleaving&mdash;an energy-efficient data layout transformation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, pp. 44:1–44:26, June 2015.
18. J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir, “A compiler framework for extracting superword level parallelism,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’12, (New York, NY, USA), pp. 347–358, ACM, 2012.
19. Intel Corporation, “Intel SPMD Program Compiler.”
20. G. Rapaport, A. Zaks, and Y. Ben-Asher, “Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 718–727, May 2015.
21. D. Molka, D. Hackenberg, R. Schne, T. Minartz, and W. Nagel, “Flexible Workload Generation for HPC Cluster Efficiency Benchmarking,” Springer Berlin / Heidelberg, 2011.
22. C. Kim, N. Satish, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, “Technical Report: Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology,” 2012.
23. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing,” in *Proc. of the 2009 IEEE Int. Symp. on Workload Characterizatio*, pp. 44–54, IEEE, 2009.
24. M. Li, R. Sasanka, S. V. Adve, Y. kuang Chen, and E. Debes, “The ALPBench Benchmark Suite,” in *In Proc. of the IEEE Int. Symp. on Workload Characterization*, 2005.