VP^3: A Vectorization Potential Performance Prototype

David C. Wong
David J. Kuck
Intel Corporation
david.c.wong@intel.com
david.kuck@intel.com

Vincent Palomares *
Zakaria Bendifallah
Mathieu Tribalat
UVSQ
{firstname}.{lastname}@uvsq.fr

Emmanuel Oseret
William Jalby
Exascale Computing Research
emmanuel.oseret@exascale-computing.eu
william.jalby@uvsq.fr

Abstract

We present VP^3, a new methodology and tool prototype for vectorization insight. It works on real applications and data, and offers accurate performance gain bounds. VP^3 guides expert developers toward the highest potential performance gain sections of the application and helps them avoid those with low potential. Choices of vectorization legality are left to the developer. Examples of its effectiveness are given. The ideas and insights provided may also be of use to compiler designers as well as system architects.

Categories and Subject Descriptors C.1.2 [*Computer Systems Organization*]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD); C.4 [*Computer Systems Organization*]: Performance of Systems—measurement techniques, modeling techniques, performance attributes; D.2.7 [*Software*]: Distribution, Maintenance, and Enhancement—enhancement

General Terms Measurement, Performance, Design, Experimentation

Keywords Dynamic and static measurement, Performance prediction, Vectorization tool, Cape, CQA, DECAN

1. INTRODUCTION

High end microprocessors rely more and more on vector hardware units and instruction sets to increase their peak performance, including VIS[1] and MMX[2] (64 bits), AltiVec[3], SSE[4] and NEON[5] (128 bits), AVX[6] (256 bits) and soon AVX-512[7] (512 bits). On AVX-512, maximum DP scalar code performance is $1/8^{\text{th}}$ of the peak. Using full vector length versus scalar results in a modest increase in power with potentially important performance gains [2], leading to major energy/performance gains, so there is a strong incentive to exploit vector units as much as possible.

To that end, much effort has been spent on automatic vectorization technology. Vectorizers detect dependences that prevent vectorization, and also try to remove harmful dependences using loop transformations such as splitting, unroll and jam, array renaming, interchange, etc. Finally, they emit vector code, dealing with target instruction set limitations. For example, non-unit stride vector loads are not supported in SSE or AVX, so the compiler applies transformations to avoid non-unit stride accesses. As a result, vectorizers have reached a very high level of complexity. In practice, advanced analyses are limited to control compilation time, so many vectorization opportunities are not exploited [10]. Furthermore, some vectorization restructuring needs are very costly or beyond automatic tools (e.g. tracing for exact data dependence analysis), so developers must hand-modify codes. All of this makes vectorization expensive, so vectorization efforts must be spent wisely.

When automatic vectorization results are disappointing, developers need simple ways to understand their reasonable options, given the available time and skills. After profiling, their best current option is often adding SIMD directives to experiment with potential performance gains. This can force compiler vectorization by ignoring data dependences. The

* Most of the work was done while at Intel Corporation

[1] SPARC® Visual Instruction Set™
[2] Intel® MMX™ technology
[3] IBM® AltiVec™
[4] Intel® Streaming SIMD Extensions™
[5] ARM® NEON™
[6] Intel® Advanced Vector Extensions™
[7] Intel® Advanced Vector Extensions 512™

generated code can reveal potential performance gains, but less robustly than VP^3, for several reasons:

1. Compilers may fail to vectorize some loops, even with directives.
2. Compiled code may crash because it is incorrect.
3. The compiler does not produce the detailed quantitative information that VP^3 can.

As a developer tool, VP^3 would be used whenever a vectorizing compiler gave weak results. VP^3 would allow developers to understand their best restructuring options, together with the potential performance payoff of each.

Both SIMD directives and VP^3 suffer from the major aggravation that potential gains depend on dynamic code properties such as loop iteration count (short vectors benefit less from vectorization than long ones) and operand locations (too many RAM accesses lead to minor payoff). Vectorization is only relevant when it can be done legally (i.e. preserving the final output) and with a satisfying speedup. VP^3 and SIMD directive insertion can both violate program semantics, and evaluating vectorization legality is very costly, so first getting an idea of potential performance gains can be an efficient way of trimming out low-incentive candidates from the list of loops to consider.

Vectorization is not an all-or-nothing activity; partial vectorization (e.g. vectorizing only the floating-point operations) can yield solid performance gains. With SIMD directives this is hard to control, whereas VP^3 fully exploits partial vectorization automatically. VP^3 has options allowing it to assume non-unit stride, gather, or scatter architectural features, and give performance estimates even where given architecture's compiler fails. Manual optimization choices can be delicate because they depend upon target architectures' vector lengths and instruction set characteristics. For example, the vector units of the recent Silvermont[8] and Haswell[9] processors are very different. VP^3 architecture options can be used to choose the best-performing system for a given code.

The major contribution of this paper is VP^3, a methodology and tool prototype, which assesses potential vectorization performance gains, guides the user toward loops with potentially high benefits, and has the following properties:

1. Quantitative assessment of vectorization performance gains, in terms of the target architecture's details, taking into account dynamic constraints such as loop iteration count and operand location.
2. Single pass, no-fail operation, in contrast with the complexities of deciding where to insert SIMD directives.
3. Practicality relative to running speed (few times real time), *in vivo* analysis of real production applications and data, and at least 80% coverage of total real time.

Section 2 describes VP^3 operation from a user's point of view, while Section 3 has examples of usage on real applications. Section 4 gives the general architecture/principles of the tool. Section 5 presents VP^3 methodology and validation of the performance gains predicted versus real measurements. Section 6 presents tool extensions under development.

2. TOOL OPERATION

2.1 General Objectives for Prediction Tools

Performance tools usually work on the principle of informing users about the current situation inside a computation. For an existing program and data set, profilers show where the execution time is spent and performance problem summaries are developed to point to problems on the basis of compiler and run-time information. Seldom does a tool *predict* performance gains in specific areas of a source code. VP^3's capabilities are to distinguish good performance from bad according to a user-defined threshold, and to make accurate predictions about performance above the threshold. It can do this for the cache-neighborhood of any data set provided for measurement, and report that neighborhood to the user (e.g. L3 contained).

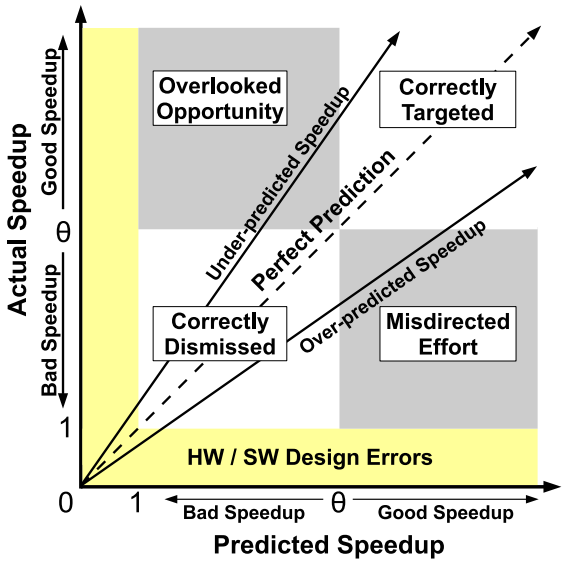

Figure 1: Operating Space of a Performance Prediction Tool

Fig. 1 shows the overall situation for any such tool, relative to performance and to the prediction quality. The axes are defined as speedup by vectorization over scalar time and are marked to show good and bad performance. The x-axis represents VP^3 predicted speedup, and the y-axis shows actual speedup obtained by compiler or manual vectorization.

[8] Intel® Silvermont microarchitecture
[9] 4th Generation Intel® Core™ processors

The point θ marked on each axis marks a threshold performance that the tool user chooses as a worthwhile vectorization performance goal. It may depend on the maximum theoretical speedup possible, the time allowed for the vectorization work, and the skill of the SW development team. The band between 0 and 1 represents various types of performance that we call erroneous, where vectorization may lead to slowdowns. Such issues have various HW and SW sources beyond the scope of this paper.

Within the graph, labels show the tool quality regions. The dashed line with slope of 1 represents perfect predictions made by an ideal tool. The four quadrants show prediction quality. We can state 3 tool objectives, in rank order:

1. Reject candidates with performance potential below threshold θ ("correctly dismissed").
2. Quantify gains for the complementary region of potential performance improvement ("correctly targeted").
3. Carry out these two types of predictions with relatively low respective errors.

The lower left quadrant is most important because it saves developers time and effort by not having to even consider certain loops. The upper right quadrant forms the target loops, and the solid lines with slopes above and below 1 denote regions with large errors, which may also be set by user parameters. If a tool projects a sufficiently large gain, and the gain obtained is even greater, few users will be unhappy. Even if the gain is less than projected but above the threshold, the tool is useful.

The lower right quadrant of "misdirected efforts" shows the region where a tool gives a high projection but the actual gain is below the threshold. The upper left quadrant of "overlooked opportunities" shows rejected loops that are actually worth vectorizing. Both quadrants are to be avoided.

2.2 Tool Output

VP^3 output for each target loop is an estimated vectorization speedup gain. Such output is augmented by an analysis of the main source of performance losses, including:

- Whether the loop performance is dominated by data access (Load Store instructions) or arithmetic (Floating Point instructions) and by how much. This is useful to guide the user in optimization and in particular through partial vectorization.
- For data access bound code, non-unit stride access is reported as is the most heavily used memory hierarchy level. More precise data on non-unit stride values can be provided at the cost of an additional run. Also, VP^3 will accept (by default) non-aligned vector loads and stores. An estimate of potential gain by alignment can also be provided.
- For Floating Point bound code, potential recurrence limitations, slow operations or code bloated by address computations or data reorganizations are reported.

This reporting can be even further refined by compiling the code with -g option enabling to establish a correlation between assembly load/store instructions and arrays in the source code.

3. EXPERIMENTAL RESULTS

3.1 Motivating Example

YALES2 [1, 11] is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. It is a finite volume code for unstructured meshes, with an innovative 4$^{\text{th}}$ order spatial scheme for the discretization of convective and diffusive terms. It is based on the low-Mach number approximations of the Navier-Stokes equations, which solves an elliptic Poisson equation at each iteration and scales well to over 16K cores. The MPI version uses subdomain decomposition with adjustable domain size, allowing efficient cache usage.

```
do ip=1,el_grp%npair
S1   ino1 = pair2node1_v(ip)
S2   ino2 = pair2node2_v(ip)
S3   co = sym_op_v(ip)*(r1_p_v(ino2) - r1_p_v(ino1))
S4   prod_r1_v(ino1) = prod_r1_v(ino1) + co
S5   prod_r1_v(ino2) = prod_r1_v(ino2) - co
enddo
```

Figure 2: YALES2 loop example

Consider vectorizing the YALES2 loop of Figure 2. Statements *S1* and *S2* obviously vectorize. *S3* is more challenging because the indirect addressing needs hardware support (vector gather instructions) which is available on the latest generation of Intel® Haswell processors. *S4* and *S5* are much more difficult because indexes `ino1` and `ino2` can take values throughout the loop execution that create dependences within *S4* and *S5* themselves, and between them. So there are 3 levels of vectorization opportunities in Figure 2.

1. The easiest to vectorize are *S1*, *S2* and *S3* and the FP operations in *S3*, *S4* and *S5*. While within autovectorizer capabilities, the current vectorizers did not vectorize.
2. The LS dependences of *S4* and *S5* due to indices `ino1` and `ino2`, can be removed by introducing coloring to ensure that `ino1` and `ino2` values are distinct. This is well beyond autovectorizer capabilities, but VP^3 can suggest, e.g. that scatter instructions would be usable if the developer introduced coloring. Forced vectorization using SIMD directives can cause the generated code to be wrong with the application producing errors or crashing.
3. To avoid all of the issues related to indirect addressing, VP^3 could advise the code developer to first use coloring to remove dependences, and second to entirely restructure the arrays to generate stride 1 accesses, which will

be then easy to vectorize. This second step is a major undertaking, requiring one to copy portions of irregular data structures into regular ones.

Solutions 2 and 3 put increasing burden on the code developer (several weeks of code rewriting). Before embarking on such efforts, developers want to know how much performance gain to expect. The main goal of VP^3 is to provide such answers. However, succeeding in vectorizing is not enough, because vectorization is highly dependent upon data access location: if data is in L1, performance gains will be much larger than if operands have to be fetched from memory. In the latter case, it means that code developers will not only have to perform data restructuring but also blocking, further increasing the cost of vectorization.

3.2 VP^3 on YALES2

Vectorizing YALES2 is a major challenge due to the indirect accesses induced by the irregular mesh structure: many of the loops have a structure very similar to the one presented in Figure 2. Of the 200 loops necessary to achieve 80% coverage of the total application time, about 2/3 are data access bound, the rest being FP bound. Some loop bodies are fairly complex; two contain more than 300 assembly instructions. On Sandy Bridge architectures, the lack of scatter or gather operations would be a performance killer for vectorization.

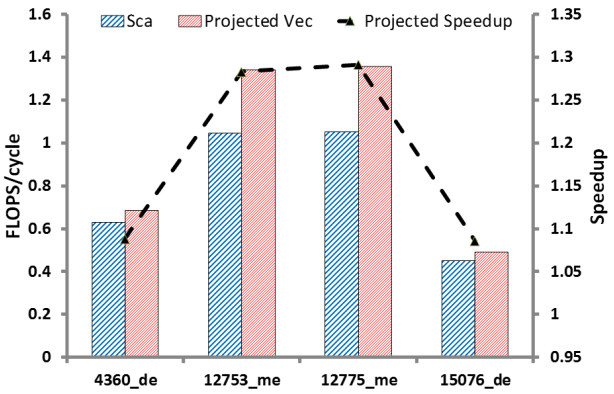

Figure 3: VP^3 Projection Results for YALES2: Low Prospects for Vectorization

For YALES2, the results shown in Fig. 3 corresponds to a Sandy Bridge target. Since Sandy Bridge does not support Scatter/Gather instructions, VP^3 was set not to perform Load/Store vectorization. The performance gains remain modest (1.2 to 1.3X). Since the cost for vectorizing this code was extremely high, and VP^3 showed limited performance gain potential, we did not push for this optimization for Sandy Bridge.

On Haswell, the loads could be vectorized but the overall gain would remain limited due to performance limitations of the gather instructions.

3.3 VP^3 on POLARIS(MD)

POLARIS(MD) (developed at CEA DSV) is a parallel code that simulates microscopic molecular systems using sophisticated interatomic potentials (including polarization effects and beyond). It includes an efficient multi-level polarizable coarse grained approach to modeling an extended chemical environment (from nanometer to micrometer scale). It is well-suited to investigate the properties of solvated protein systems and of heavy ions in complex chemical environments [15].

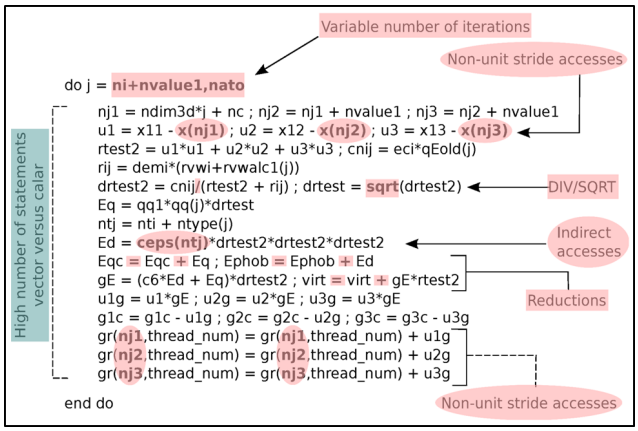

Figure 4: Source Code for POLARIS Loop Example: Out of all the potential performance issues, only the division and square root operations impede performance.

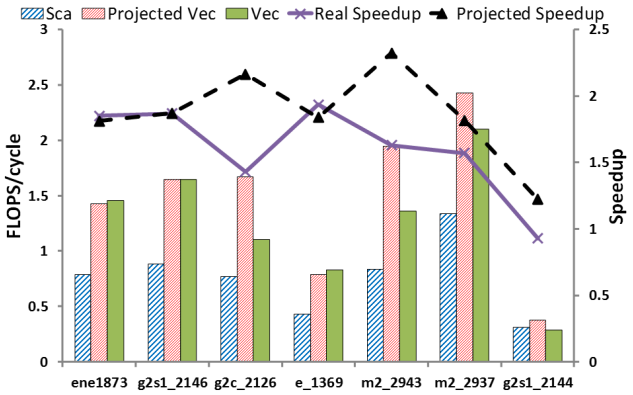

Figure 5: VP^3 Projection Results for POLARIS: VP^3 vs. Measurements

A typical loop computing interactions between pairs of atoms is shown in Fig. 4. This loop was originally not vectorized to preserve roundoff. The key issue was to determine what was worth vectorizing. A first VP^3 run showed that the cost of FP instructions was 4 times higher than the cost of Load Store instructions; the effort had to be focused on FP vectorization. VP^3 further revealed that partial vectorization of FP would provide a 2X performance improvement simply due to the SQRT and DIV vectorization. SIMD directives

were added, full runs were made to check that round off errors did not change, and the loop got a 2X speedup. The code contained 40 similar loops (i.e. containing DIV/SQRT) for which VP^3 gave similar diagnostics. The SIMD directive was applied to all 40 loops resulting in a 1.5X speed up on the whole application.

VP^3 was run on 7 fairly large POLARIS codelets (up to 111 assembly instructions in the loop body), including the FP saturated one discussed above. Fig. 5 shows the scalar (Sca), projected vector (Projected Vec) and hand-vectorized (Vec) results, as well as their speedup ratio (Real Speedup).

One of them actually shows an unforeseen slowdown due to unexpectedly complex address computations, causing a 25% over-prediction. Furthermore, it is in the lower right quadrant in Fig. 6: the user would be misled into vectorizing it. The 6 others are correctly estimated to highly profit from vectorization. Despite 2 being over-predicted by around 30%, the user's minimum expectation would be met.

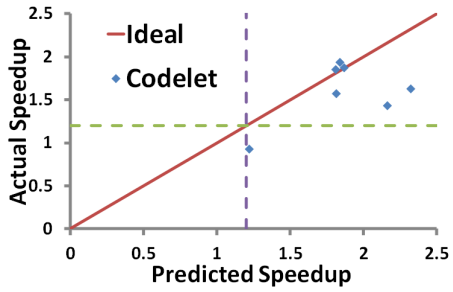

Figure 6: Operating Space of VP^3 on POLARIS ($\theta = 1.2$)

4. TOOL PRINCIPLES

The key principles of the method are to first isolate the performance contributions of FP (floating-point) arithmetic operations and LS (load/store) data access operations, then model the performance impact of vectorization on each of these components and finally recombine them to get the global performance.

Starting with a scalar binary loop, vector performance (in cycles/iteration) prediction proceeds in 4 phases (see Fig. 7):

1. FP/LS variants generation and measurement (Section 4.1):
   (a) Generation of FP and LS variants of the original scalar loop using DECAN [9].
   (b) Execute and measure the FP and LS variants, as well as the original binary.
2. Static projection of FP and LS vectorized performance using the generated FP/LS variants (Section 4.2).
3. Refinement of projected LS vectorized performance with memory traffic information gathered from scalar execution to account for memory hierarchy performance bottlenecks that limit vectorization benefits (Section 4.3).
4. Combine the projected FP and LS vectorized performance to get overall vectorized performance (Section 4.4).

The vector performance can be refined further by adding extra time for compiler-generated peel/tail loops[10] .

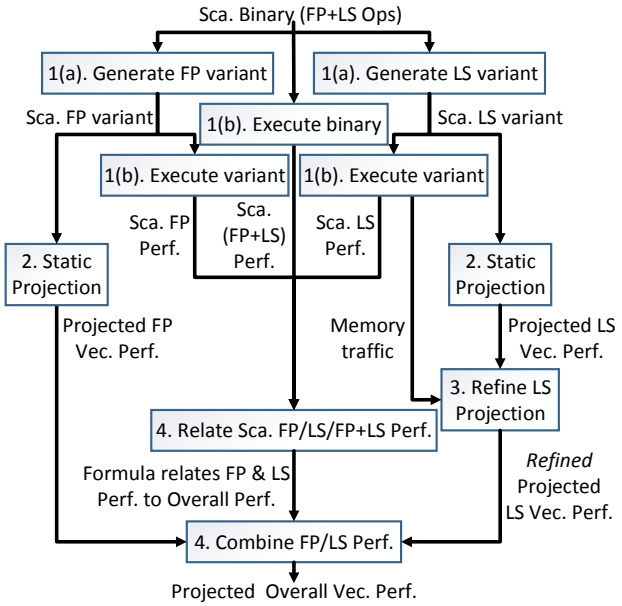

Figure 7: VP^3 Vec. Projection Steps

4.1 FP/LS Variants Generation and Measurement

Starting with a scalar binary loop, VP^3 invokes DECAN [9] (a binary loop transformer) to generate two scalar binary variants: LS (in which all FP arithmetic instructions have been suppressed) and FP (in which all data access instructions have been replaced by `XORPS` instructions on the same register in order to avoid introducing extra dependences).

These scalar variants, along with the original binary loop, are executed with different input data sizes to explore memory accesses from L1, L2, L3 and RAM. For each execution, in addition to measuring cycles per iteration, performance evaluation counters are used to collect memory traffic information (less than 10 counters are required to model data traffic accurately between all cache levels). These dynamic measurements are used to refine LS projection (Section 4.3).

4.2 Static Projection of FP/LS Operations

This phase is entirely built around CQA [4], a tool for analyzing binaries statically. CQA estimates a binary loop's performance (cycles/iteration) without executing it, by assuming all operands are in L1 and the loop trip count is infinite.

For each scalar FP/LS binary loop generated in Section 4.1, CQA can project the vectorized performance by generating a vectorized version of the corresponding scalar

[10] These loops are generated to get proper data alignment for vectorized loop execution and deal with leftover iterations.

binary loop (called vector mockup code) and statically analysing the vectorized loop.

To generate the vector mockup code for a loop, CQA unrolls and jams consecutive iterations to regroup the unrolled instances of scalar instructions into "packs", which become candidates for vectorization. The jamming process is made taking the dependence into consideration. CQA examines each pack of scalar instructions and tries to replace it by vector equivalents using the same register operands. There are three kinds of scalar instruction packs:

1. Simple scalar FP instructions (e.g. `ADDSS` instruction with only register operands) are simply replaced by their vector equivalent (e.g. `ADDPS`).
2. Simple scalar Load/Store instructions (e.g. `MOVSS` instruction with a memory operand) are replaced by a vector equivalent (e.g. `MOVAPS`), if CQA detects a stride-1 memory access pattern. Otherwise, for non-unit stride accesses, the instruction is not vectorized and special instructions (insert/merge) are added to pack scalar operands into a vector register.
3. Scalar FP-LS mixed instructions (e.g. `ADDSS` instruction with a memory operand) are first split into a simple FP and a simple Load/Store instruction. These instructions are then handled as described in 1 and 2 above.

Sometimes the user may want VP^3 to project the vectorization performance when *only FP* operations are vectorized. This is done by generating the insert/merge instruction even when the Load/Store instruction performs stride-1 access.

4.3 Refinement of Static Projection of LS Operations

To refine the vector performance of LS Operations to account for performance degradation due to L2, L3, RAM and TLB activities,

VP^3 uses Cape [12] with inputs from CQA static analysis (Section 4.2) and dynamic measurements from Section 4.1. Cape projects *cycles per iteration* of a loop by modeling a system consisting of N nodes. Each node abstracts a hardware subsystem using software-related metrics. A maximum operation rate called $bandwidth$ is associated with each node (B_i for node i). Per loop iteration, node i will perform O_i operations. Cape uses Equation 1 to compute the *cycles per iteration* T of the loop. To get the vector performance

$$T = \max_{1 \le i \le N} \frac{O_i}{B_i} \quad (1)$$

of LS operations, the relevant nodes are:

1. L1 Load, L1 Store, front-end, issue ports. The operation counts are obtained from CQA analysis results. The microarchitecture bandwidths are obtained from [8].
2. L2, L3, RAM. The operation/traffic counts are obtained from performance counters (Section 4.1). Here, we assume the data access pattern of a vectorized loop is the same as the scalar version of the loop, so we can use operations counts from scalar loop runs. The bandwidths are obtained by using scaling factors from Table 1 to scale up the bandwidths determined from scalar loop runs. The scalar bandwidths are empirically determined by observing the peak traffic rate when the scalar loop is executed with different input data sizes.
3. TLB. The operation count is obtained from a performance counter, multiplied by a TLB miss penalty. The bandwidth is assumed to be 1. Similar to L2/L3/RAM node, we assume the TLB behavior of a vectorized loop is the same as the scalar loop.

Table 1: BW Scaling Factors (BW Vector / BW Scalar)

Type of Scalar	L2	L3	RAM
Single Precision	2.04	1.63	1.25
Double Precision	1.79	1.40	1.10

In bullet 2 above, Table 1 was used for bandwidth scaling for SSE vector instructions. It is measured by running various kernels using different types of vector instructions (loads/stores, SP/DP). Numbers of streams were measured, and compared with equivalent kernels using scalar instructions. Table 1 shows that vector instructions nearly double the bandwidth of L2 but the increase drops when accessing the farthest levels (L3 and RAM).

4.4 Combining FP/LS Projection Results

With performance projection for FP (T_{FP}, Section 4.2) and LS (T_{LS}, Section 4.3) in cycles per iteration, VP^3 combines them to generate the overall performance estimate (T_{REF}). VP^3 assumes the relationship between T_{FP}, T_{LS} and T_{REF} can be related by a *fitting* function f such that $T_{\text{REF}} = f(T_{\text{FP}}, T_{\text{LS}})$. Therefore, to compute overall performance projection, VP^3 applies f with projected performance of FP and LS as inputs. The function is continually fitted by using *scalar* loop measurements described in Section 4.1 where scalar performance of FP, LS and overall are measured. VP^3 assumes the same function can be applied for both *scalar* and *vector* versions of the loop.

4.5 Tool Speed

High time-consuming tools can only be useful to find a few hot loops. VP^3 needs no trace and requires only several runs of the instrumented program, as follows. First, generation and execution of microbenchmarks for Table 1 is made once for all applications running on the same target architecture and is therefore negligible. Second, CQA passes (Section 4.2) and Cape (Section 4.3 and 4.4) computations are extremely fast (under a few seconds). So the cost of VP^3 is low enough to deal with a few hundred loops per application. This is often essential because many real-life applications have a very flat profile; reaching 80% coverage of total execution time may require 200 loops.

Most time consuming for VP^3 is application-specific measurements described in Section 4.1. A static analysis CQA pass allows the identification of unvectorized (or partially vectorized) target loops. Then running the application provides information about target loops, e.g. their weight in the program's total execution time and iteration counts. This can be done using Intel compilers, which provide max, min, and average, at modest overhead cost (20%), or using the MIL infrastructure [3] to provide more detailed information on the loop iteration count distribution at the cost of a higher overhead. Gathering traffic information via counters requires 3 further runs. Typically, the overhead associated with such collection remains under 20%. Next we execute the FP and LS DECAN variants. This can be achieved in a single run by using a rollback mechanism after each target loop; the overhead is 2X the execution time coverage. In total, 5 application runs are needed, with various slowdowns. Assuming a target coverage of 80%, the total time is around $1.2 + 3 \times 1.2 + (1 + 2 \times 0.8) = 7.4$ full application runs.

5. VALIDATION

5.1 Methodology, Measurements and Experimental Settings

To assess the quality/limitations of VP^3, we picked a set of loops which were vectorized using a standard vectorizer. Then we forced the compiler to generate scalar versions using proper flags. These scalar versions were fed into VP^3 to generate estimates of potential vectorization gains. These estimates were compared with measurements of the vectorized loops.

The loops used for validation were extracted from Numerical Recipes (NR) [13] and previously used to validate Cape's features [12]. They were carefully selected to cover a wide range of different behavior: 1D/2D loops, 1D/2D arrays, unit and non-unit stride accesses. Both vector and scalar variants were compiled using Intel® Fortran Compiler (v12.1.3), using compiler flag "-O3" (as well as "-no-vec" for the scalar versions of the NRs, to prevent auto-vectorization). We made a few tests with compiler version v14 but (in most cases) differences were minimal.

All measurements were performed on a quad-socket Intel® Xeon E5-4640 (Sandy Bridge[11]) with Intel® Hyper-Threading Technology deactivated, on RHEL 6.3 x86-64 with Transparent Huge Pages . On every CPU, each of the 8 cores has dedicated: L1 instruction cache of 32KB, L1 data cache of 32KB and L2 cache of 256KB, plus access to a 20MB L3 cache shared by all cores of the chip.

5.2 Error Analysis

To capture the expected vectorization *benefits/effort*, VP^3 allows developers to set threshold θ (see Fig. 1), to focus only on loops for which potential vectorization performance gains are above threshold. As an example, we set default $\theta = 1.2$, but for a given type of app and developer skills, any value may be chosen. Fig. 8 summarizes, for the 16 validation codelets, the number of errors (mispredictions) and the breakdown between over predictions and under predictions. We allow a default tolerance of ± 0.02, i.e. when the user asks for $\theta = 1.2$, we guarantee a prediction for $1.18 \leq \theta \leq 1.22$. Note that the number of errors decreases linearly to 1 as the tolerance goes to 0.03. Developers could adjust the tolerance to study error sensitivity for a given set of apps, before doing their work.

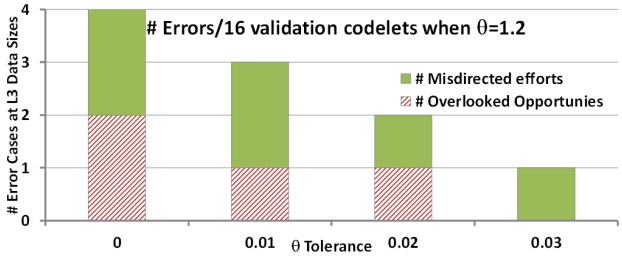

Figure 8: Error Cases/16 Validation Codelets vs. θ Tolerance

Our performance prediction methodology has the following sources of deviations from compiler vectorization results (cases 1. and 2.) and potential errors (3.):

1. **Dependence violation**: during the mockup vector generation in CQA, vectorized statements for which there are RAW dependences between memory accesses could exist. Since we do not have such dependence information, our vectorization process could generate incorrect code and therefore an erroneous (conservative) prediction. Other types of dependence (WAR or WAW) are not an issue since a powerful vectorizer can easily work around them to generate vector code.
2. **Poor scalar code**: since the VP^3 vectorization process is a line by line replacement of scalar by vector instructions, it is very sensitive to the code quality of the initial scalar code. This is why we explore several unroll variants (via directives) to ensure that our process uses a high quality starting point. Note that VP^3 will not alter cache traffic. This is acceptable because apart from some corner cases, cache-optimization transformations are profitable to both scalar and vector versions. Therefore, if loop interchange is beneficial to scalar code, our starting point should be a loop interchanged scalar binary.
3. **Load-store variation**: to account for the increase in performance over scalar code for vector data access, we use a simple scaling table obtained by microbenchmarking (Tab. 1). For the moment, this table does not distinguish between loads and stores and the number of streams (separate array accesses). This can be improved by using more extensive benchmarks, and static analysis which will give us the number of load and store streams.

[11] 2nd Generation Intel® Core™ processors

6. EXTENSIONS

In this paper, we have assumed that loops do not contain any branches. Recent vector instruction sets support masked operations which obtain acceptable vector efficiency for loop structures containing simple branch structures, i.e. (if…then…else). Our framework can be extended to cover such cases by using static analysis along the two potential paths. The mockup generation in CQA will assume that the two paths are generated using masked instructions.

The dynamic analysis then has the extra task of modeling the branch outcome time accounting for the branching probabilities. Assuming that traffic remains unchanged, the LS prediction can be performed in a similar manner to loops without branches. This is a valid assumption if the branch alternates targets at a fine level. Otherwise, a different method would have to be used.

Besides predicting the impact of moving from scalar to vector, VP^3 can also be used for moving from SSE to AVX, from AVX to AVX-512, or any pair of similar vector instruction sets. Some care has to be taken when moving among architectures because cache size changes may alter traffic.

7. RELATED WORK

Due to the renewed importance of vector units, a lot of research effort has been recently devoted to vectorization. First, [10] analyzed in depth the major deficiencies of current vectorizers and proposed to (re)incorporate classical optimization techniques to improve automatic vectorization process. Then, to improve classical static dependence analysis, Vector Seeker [5] used on-line dynamic memory tracing analysis to obtain an accurate dependence analysis. The on-line tracing process is costly but it can detect candidate loops for vectorization which could not be analyzed satisfactorily with classic static techniques. Holewinski [7] took a similar angle but restricted their effort to regularly indexed loops. However, they provided useful information on loops for which unit-stride access could be obtained therefore improving the vectorization process. Finally, [14] propose combining static information (compiler optimization reports detailing the main reasons why the compiler failed to vectorize) with dynamic information (loop coverage, iteration count, stride, alignment) to help the user in selecting target loops for vectorization. Haine [6] tried to vectorize directly at the assembly level, and therefore is subject to the same issues as SIMD directives.

None of the above provides estimates of vectorization performance gains. VP^3 is in fact complementary to the previous work. Running VP^3 first will allow the selection of loops of interest, and then tools like Vector Seeker can be used to discover whether or not vectorization is legal.

8. CONCLUSIONS

We have presented our fast and flexible prototype VP^3 for vectorization insight. VP^3 can be used by compiler and software developer to focus on loops with high vectorization gain, taking into account dynamic constraints like loop iteration count and operand location. To validate VP^3, we used 16 validation codelets extracted from [13] and [10]. We observed that VP^3 can project the vectorized performance from scalar performance within reasonable error thresholds. We also performed further investigations and have identified sources of errors, which will drive our future improvements of the tool. In addition to evaluating VP^3 with respect to difference between projected and measured vectorized performance, we also evaluated the results considering the impact to the user. To evaluate VP^3 on real world problems, we used VP^3 to analyze POLARIS and YALES2. For POLARIS, the results matched well the previous human optimization effort. We believe VP^3, as it stands, is a useful tool for those who want to improve software performance by vectorization. With the improvement we are performing, VP^3 will deliver higher quality insight to the users.

References

[1] YALES2 public page. URL `http://www.coria-cfd.fr/index.php/YALES2`.

[2] J. M. Cebrián, L. Natvig, and J. C. Meyer. Performance and energy impact of parallelization and vectorization techniques in modern microprocessors. *Computing*, 2013.

[3] A. Charif-Rubial et al. MIL: A language to build program analysis tools... HiPC, 2013.

[4] A. S. Charif-Rubial et al. CQA: A code quality analyzer tool at binary level. *to appear in HiPC '14*.

[5] G. C. Evans et al. Vector seeker: A tool for finding vector potential. WPMVP '14.

[6] C. Haine et al. Exploring and evaluating array layout restructuration for SIMDization. *to appear in LCPC '14*.

[7] J. Holewinski et al. Dynamic trace-based analysis of vectorization potential of applications. *SIGPLAN Not.*, 2012.

[8] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-030. Sept. 2014.

[9] S. Koliaï et al. Quantifying performance bottleneck cost through differential analysis. ICS '13.

[10] S. Maleki et al. An evaluation of vectorizing compilers. PACT '11.

[11] V. Moureau et al. From large-eddy simulation to direct numerical simulation of a lean premixed swirl flame: Filtered laminar flame-pdf modeling. *Combustion and Flame*, 2011.

[12] J. Noudohouenou et al. Simsys: A performance simulation framework. RAPIDO '13.

[13] W. H. Press et al. Numerical recipes in C: The art of scientific computing. second edition, 1992.

[14] A. Rane et al. Unification of static and dynamic analyses to enable vectorization. *to appear in LCPC '14*, 2014.

[15] F. Réal et al. Further insights in the ability of classical non-additive potentials to model actinide ion-water interactions. *Journal of Computational Chemistry*, 2013.