A Code Generation Approach for Auto-Vectorization in the SPADE Compiler

Huayong Wang[1], Henrique Andrade[2], Buğra Gedik[2], and Kun-Lung Wu[2]

[1] IBM China Research Lab,
huayongw@cn.ibm.com,
[2] IBM T. J. Watson Research Center,
{hcma,bgedik,klwu}@us.ibm.com

Abstract. We describe an auto-vectorization approach for the SPADE stream processing programming language, comprising two ideas. First, we provide support for vectors as a primitive data type. Second, we provide a C++ library with architecture-specific implementations of a large number of pre-vectorized operations as the means to support language extensions. We evaluate our approach with several stream processing operators, contrasting SPADE's auto-vectorization with the native auto-vectorization provided by the GNU `gcc` and Intel `icc` compilers.

1 Introduction

Stream processing applications are designed to identify new information from data that is continuosly generated by software and hardware sensors [1, 2]. Stream processing has been a very active area of research over the last few years. Frameworks such as STREAM [3], Borealis [4], StreamBase [5], TelegraphCQ [6], among others, have focused on providing stream processing middleware and declarative language for writing applications. Languages such as StreamIt [7] and the Aspen language [8] provide additional support and abstractions for simplifying the coding of stream processing applications. Another active research area has been in investigating the benefits of vectorization. From fundamental operations such as matrix multiplication [9] to complete analytics in areas such as image processing [10], multimedia applications [11], and digital signal processing [12], it has been shown that substantial performance improvements can be obtained. Drawing from these two research lines, our contributions in addressing the challenge of speeding up stream processing applications can be summarized as follows: (1) an auto-vectorization approach that relies on a two-tier programming model exposed by the SPADE language; and (2) an empirical case study with real-world stream processing operators, demonstrating the improvements that can be derived through the auto-vectorization support provided by the SPADE compiler.

2 Auto-Vectorization Support in SPADE

System S [1, 2] is a large-scale, distributed data stream processing middleware. It supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes. System S applications are written in the SPADE(Stream

Processing Application Declarative Engine) programming language [13, 14]. The rest of this section describes how we added auto-vectorization capabilities to the SPADE compiler.

In general terms, vectorizing computational kernels is usually accomplished by one of the following alternatives: (1) By directly using SIMD instructions, i.e., by writing the code with vectorization instructions either by inlining the assembly language instructions into the source code or through the use of *intrinsics* [15]; (2) By employing specialized libraries that implement common domain-specific operations [12] required by the data processing analytics; or (3) By making use of general-purpose auto-vectorizing compilers [16] where compiler analysis is used to identify vectorization sites in the code.

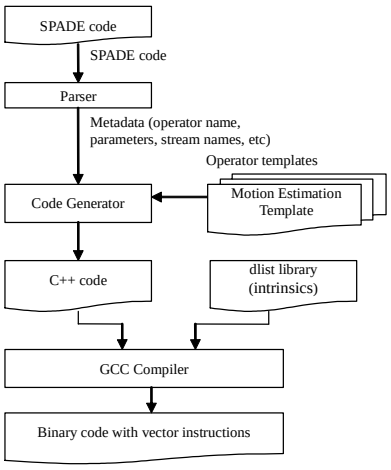


Fig. 1. The two-tier code generation framework. The Motion Estimation operator is discussed in Section 3

Our approach, summarized in Figure 1, borrows from all of these techniques. In a nutshell, it provides direct auto-vectorization capabilities to SPADE language programmers and the means for relying on pre-vectorized building-block operations, thus simplifying the task of extending the language for SPADE operator programmers. This approach consists of exposing a two-tier programming model to application developers.

The application composition programming tier. SPADE is an extensible, distributed application composition language. A typical application consists of an *enumeration* of streams produced by a flowgraph. Each stream is defined in terms of its schema, i.e., the format for the tuples it transports. In such a flowgraph, each stream is the result of applying an *operator* that carries out a particular data transformation.

Figure 2 shows a simple exemplar application, with a contrived credit card transaction fraud detection application. We refer to this programming tier as the *application composition* programming tier. Most of the application developers do the bulk of their coding in this tier, by employing operators available from existing operator toolkits. In this tier, a developer employs the SPADE language syntax and constructs. In this context, several low-level data transformation operations can be carried out on vectors, which are basic data types supported by the language. In this tier, auto-vectorization is achieved by the compiler generating code that converts vector manipulation expressions (e.g., the expression with `normalizedTransactionFV` in the sample code) written in the SPADE language into calls to a hand-vectorized C++ template class, called `dlist`. Currently, the `dlist` template class has two instantiations, a scalar one which is architecture-agnostic and a vectorized one, which has been specialized for x86 processors through the use of SSE intrinsics. It includes the implementation of many operations, from simple arithmetic operations with vector operands, to more specialized

```
composite FraudDetection {
    graph
     stream<timestamp transactionTime, string userid,
        list<float32> transactionFV>
      transactions = TCPSource() {
        param url: "stcpns:///CreditCardTransactions";
      }
      stream<timestamp transactionTime, string userid,
        list<float32> normalizedTransactionFV>
      normalizedTransactions = Functor(transactions) {
        output normalizedTransactions:
            normalizedTransactionFV = normalize(4*transactionFV);
      }
      stream<string userid, int32 clusterid>
      classifiedTransactions = KNN(normalizedTransactions) {
        param metric: cosineDistanceMetric
      }
      () = FileSink(classifiedTransactions) {
        param url: "file:///classifiedTransactions.dat";
      }
}
```

Fig. 2. The SPADE application composition programming tier

operations such as adding a vector and a scalar operand, to vector element filtering, plus a large collection of other convenience functions, such as the `normalize` function seen in Figure 2.

When a SPADE application runs, it does not use any form of introspection. Instead the SPADE compiler *consumes* an application source code of the type seen in Figure 2, producing the equivalent C++ code. In other words, the SPADE compiler employs a collection of backend code generators, one per operator (e.g., `TCPSource`, `Functor`, `KNN`, `FileSink` in Figure 2) in the application. Each operator code generator is tasked with specializing the template code associated with an operator's code generator based on the operator parameter configurations as well as environmental configurations. As a final step, the C++ code is compiled using a regular off-the-shelf C++ compiler.

The toolkit programming tier. The second programming tier, normally not seen by application developers, is the *toolkit programming* tier. The SPADE language does not pre-define a set of operators. New operators can be added to the language, seamlessly extending its syntax. In this case, the vectorization support is provided to toolkit writers in terms of a *mixed-mode* programming environment. This templatized programming mode combines a scripting language (in our case, Perl) for coding the generic parts of the operator logic and a regular programming language (in our case, C++) for the fixed logic. The templatized implementation is automatically converted to a code generator by the SPADE support tooling. Since vectors are first-class types in the SPADE language,

when an operator such as the KNN operator used by the application in Figure 2 makes use of vector operations, these operations are coded by making use of methods provided by the dlist template class.

3 Case Study Applications

We implemented the following vector-heavy operators to demonstrate and experiment with our auto-vectorization approach:

K-Nearest Neighbor (KNN). The KNN algorithm [17] classifies an object based on knowledge gleaned from a set of training objects. The pre-classified training set and the unclassified objects are represented as feature vectors in a multidimensional feature space. KNN can leverage vectorization in the step of computing the similarity metric, represented as the distance between the feature vector of an unclassified object and each of the training objects. In our study, we employed two of the commonly used similarity metrics – cosine and weighted cosine.

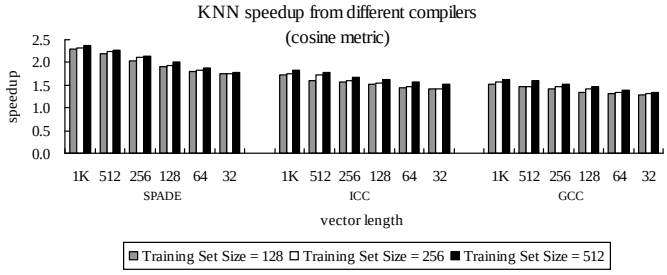
Digital Watermarking. Digital watermarking is a commonly used technology in copyright protection systems. The particular implementation we chose to study comes from the Digital Watermarking Open Source Project [18]. We implemented the Discrete Hartley Transform (DHT)-based watermarking algorithm as a SPADE operator.

Motion Estimation. Motion estimation is used in video encoding and surveillance applications [19]. Motion estimation is the process of determining object movement in adjacent video frames. In this work, we chose to implement the *full search block matching* algorithm as a SPADE operator. For our purpose, full search also makes the amount of computation fixed irrespective of the processed image frames, simplifying the experimental evaluation carried out in Section 4. We use the Sum of Absolute Differences[3] (SAD) method for movement estimation, a simple yet effective metric. It works by taking the absolute value of the difference between each pixel in the original block and the corresponding pixel in the block used for comparison. These differences are summed up, creating the similarity metric value. The excerpt below is from a dlist helper function implementing such operation:

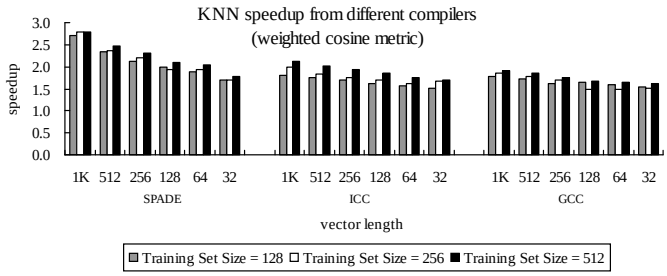
```
inline int DLIST_SAD(const dlist<unsigned char>& a, const dlist<unsigned char>& b,
                     int pos_a, int pos_b, int len)
  __m128i* p = static_cast<__m128i*>(&a[pos_a]);
  __m128i* q = static_cast<__m128i*>(&b[pos_b]);
  int res=0;
  for (unsinged i = 0; i <= len - 16; i += 16)
    __m128i va, vb, vsad;
    va = _mm_loadu_si128(p);
    vb = _mm_loadu_si128(q) ;
    vsad = _mm_sad_epu8(va, vb);
    res += _mm_extract_epi16(vsad, 0) + _mm_extract_epi16(vsad, 4);
    p++; q++;

  for (unsigned i = 0; i < len; i++)
    res += abs(static_cast<int>(a[pos_a + i]) - static_cast<int>(b[pos_b + i]));
```

[3] Note that the SSE instruction set includes the PSADBW instruction, which efficiently performs this type of computation.



(a) Cosine measurement



(b) Weighted cosine measurement

Fig. 3. KNN speedup

4 Empirical Evaluation

To assess the effectiveness of our two-tier auto-vectorization technique, we designed experiments using each of the SPADE operators from Section 3. We employed 3 experimental configurations: (1) **Scalar**: each operator was implemented using methods from the `dlist` class that did not use *intrinsics* and the SPADE compiler used `gcc` as the backend C++ compiler, with auto-vectorization turned off; (2) **C++ auto-vectorization**: the operators from the *scalar* configuration were compiled with the C++ auto-vectorization feature turned on. In this case, the SPADE compiler could be configured to employ either `gcc` or Intel's `icc` [20] as the backend C++ compiler. The results for a particular backend C++ compiler are labeled with the name of the compiler; and (3) **SPADE auto-vectorization**: each operator was implemented using methods from the `dlist` class written using the `gcc` SSE *intrinsics*. The SPADE compiler was configured to use `gcc` as the backend C++ compiler. The results for this approach are labeled with the word "SPADE".

In our empirical evaluation, we measured the *throughput* of the benchmark applications by running each configuration 5 times and averaging the results. The ratio between the observed throughput of one of the auto-vectorization approaches (either the C++ compiler-based or SPADE's) to the *scalar* version is the speedup we report. Note that, with this experimental setup, we *are* capturing the full spectrum of processing carried out by the processing operator, including the interaction between the operator and the source and sink edge adapters (responsible for data ingestion and results generation, re-

spectively), which typically do not benefit from auto-vectorization techniques. The experiments ran on a node with an Intel Core2 Duo Processor 6700 running at 2.66 GHz, with 32 KB L1 data cache per core, shared 4 MB L2 cache, and SSE3 support, running Linux. We employed `gcc` version 4.3 and `icc` version 11.0.

KNN. Figures 3(a) and 3(b) show the speedup curve as a function of the feature vector length for KNN, when using the *cosine* and the *weighted cosine* as the similarity metric, respectively. As expected, the more elements in the vector, the larger is the speedup. Also, the weighted cosine measurement has better speedup characteristics since it carries out more vector operations compared to the simpler algorithm employing the cosine measurement. We can see that, in most cases, more speedup is observed for larger training set sizes, as, in general, more vector-heavy work must be carried out. The exception is for smaller vector lengths when using the `gcc` compiler and the weighted cosine metric, as is seen in Figure 3(b).

A more interesting observation is that all auto-vectorization approaches are beneficial. In general, the SPADE auto-vectorization mechanism outperforms the `icc` Intel compiler, and the Intel compiler outperforms `gcc`. Looking at a single data point, 1K-vectors using the weighted cosine metric, we can see that the difference can be quite substantial. The SPADE auto-vectorization mechanism produces a speed-up of around 3, while `icc` yields a speedup of around 2, which is also what `gcc` is able to obtain.

Digital Watermarking. In Figure 4 we see the speedup in throughput (in this case, measured in terms of frames/s) as a function of different block sizes for two different image sizes. When the image size changes from 32×32 to 512×512, the speedup goes from 1.65 to 2.46 with the SPADE auto-vectorization mechanism. As expected, when larger images are used, the execution time for the matrix operations becomes a larger relative share of the total processing per frame. Consequently, the speedup also increases when comparing the vectorized implementation against the scalar one.

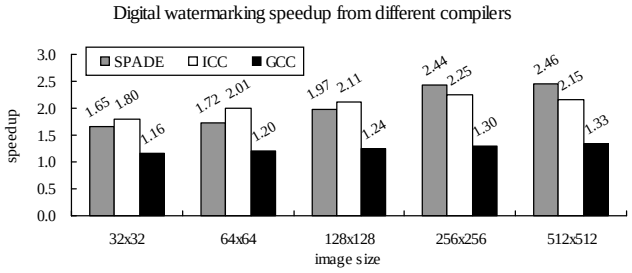


Fig. 4. Digital watermarking speedup

Note that the `icc` auto-vectorized version initially outperforms SPADE and, in general, it also outperforms `gcc`. Again, we point out that the SPADE-generated vectorized code is also compiled with `gcc` in these experiments. Thus, it can be seen that the SPADE auto-vectorization mechanism adds a factor between 0.49 and 1.13 of additional speedup, eventually, gaining on the `icc` compiler. Ultimately, for 512×512 images, the SPADE auto-vectorized version outperforms `icc`'s by an extra 0.31 factor in speedup.

Motion Estimation. To study the Motion Estimation operator's performance characteristics, we employed 256×256 video frames. The macro-block size ranged from 4×4 to 32×32. The results shown in Figure 5 demonstrate that the larger the macro-block size is, the larger is the speedup. This is because using larger macro-blocks increases

the fraction of the overall computation performing operations on vector operands. The speedups in the SPADE auto-vectorized version for different macro-block sizes are 1.82, 2.34, 3.84, and 6.44 respectively, in correlation with the sizes of the macro-blocks.

Several surprising results can also be observed. For example, the speedup can be significantly higher than usual because, using the SPADE auto-vectorization mechanism. The `SAD` operation intensively used by this operator can be computed with a single instruction when using the vectorized `dlist` class (see Section 3). Synthetic experiments, omitted for lack of space, demonstrated that `SAD` operations can be vectorized *very* efficiently (with a speedup varying from 7 to 34, under different conditions). Here we also see the largest difference between the SPADE auto-vectorized version and `icc`'s. Simply put, `icc` did not appear to recognize and employ the appropriate SSE instruction for the `SAD` operation. The same is true for `gcc`. This observation highlights why the two-tier approach used in SPADE is useful. While, we spent the effort in re-implementing the `dlist` template class with *intrinsics*, the performance payoff can be substantial. From an operator developer's standpoint, this is completely transparent, since one can directly use the `dlist` methods when implementing any vector-heavy operator.

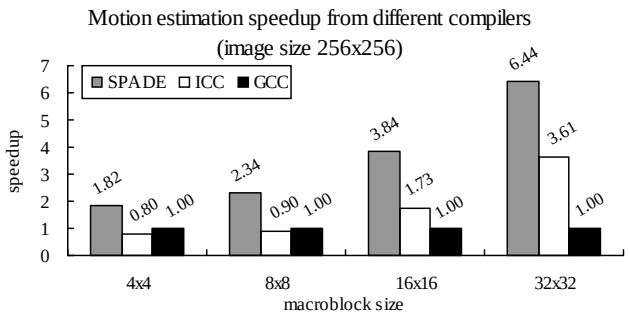


Fig. 5. Motion estimation speedup

5 Concluding Remarks

We have observed that both `gcc` and `icc` were, in most cases, successful in auto-vectorizing the code. Not surprisingly their effectiveness depended on being able to identify loop patterns amenable to optimizations. On the other hand, our two-tier approach based on providing a library with all of the basic vector manipulation operations proved successful, in most cases, substantially outperforming the direct use of the auto-vectorization capabilities provided by `gcc` and `icc`. Currently, substantial work in the SPADE compiler is focused on developing techniques for leveraging architecture-specific features of modern processors. The SPADE compiler architecture and its use of code generation makes the SPADE language specially suitable to writing high performance, large-scale, distributed, complex streaming applications.

References

1. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: Proceedings of the International Conference on Management of Data (SIGMOD'06), Chicago, IL (2006)

2. Wu, K.L., Yu, P.S., Gedik, B., Hildrum, K.W., Aggarwal, C.C., Bouillet, E., Fan, W., George, D.A., Gu, X., Luo, G., Wang, H.: Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In: Proceedings of the International Conference on Very Large Data Bases Conference (VLDB'07), Vienna, Austria (2007)
3. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: The Stanford stream data manager. IEEE Data Engineering Bulletin **26** (2003)
4. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: Proceedings of Conference on Innovative Data Systems Research (CIDR'05), Asilomar, CA (2005)
5. StreamBase: StreamBase Systems. `http://www.streambase.com`
6. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proceedings of Conference on Innovative Data Systems Research (CIDR'03), Asilomar, CA (2003)
7. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Proceedings of the 2002 International Conference on Compiler Construction (ICCC'02), Grenoble, France (April 2002)
8. Upadhyaya, G., Pai, V.S., Midkiff, S.P.: Expressing and exploiting concurrency in networked applications with Aspen. In: Proceedings of the 2007 Symposium on Principles and Practice of Parallel Programming (PPOPP'07). (2007)
9. Aberdeen, D., Baxter, J.: Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions. Concurrency and Computation: Practice and Experience **13** (2001) 103–119
10. Conte, G., Tommesani, S., Zanichelli, F.: The long and winding road to high-performance image processing with MMX/SSE. In: 5th International Workshop on Computer Architectures for Machine Perception (CAMP'00), Washington, DC, USA (2000) 302
11. Diefendorff, K., Dubey, P.K., Hochsprung, R., Scales, H.: AltiVec extension to PowerPC accelerates media processing. IEEE Micro **20**(2) (2000) 85–95
12. Puschel, F.F.M.: Short vector code generation for the discrete fourier transform. In: Proceedings of the International Conference on Parallel and Distributed Systems (IPDPS'03), Nice, France (2003)
13. Gedik, B., Andrade, H., Wu, K.L., Yu, P.S., Doo, M.: SPADE: The System S declarative stream processing engine. In: Proceedings of the International Conference on Management of Data (SIGMOD'08), Vancouver, Canada (2008)
14. Hirzel, M., Andrade, H., Gedik, B., Kumar, V., Rosa, G., Soule, R., Wu, K.L.: Spade – language specification. Technical Report RC24830, IBM Research (2009)
15. Intel: Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/design/processor/manuals/248966.pdf` (November 2007)
16. Naishlos, D.: Autovectorization in GCC. In: Proceedings of the GCC Summit. (2004)
17. Wu, X., Kumar, V., Quinlan, J.R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Yu, P.S., Zhou, Z.H., Steinbach, M., Hand, D.J., Steinberg, D.: Top 10 algorithms in data mining. Knowledge and Information Systems **14**(1) (January 2008) 1–37
18. Meerwald, P.: Digital watermarking project. `http://www.cosy.sbg.ac.at/~pmeerw/Watermarking`
19. Laidi, K., Bailiche, M.A., Mehenni, M.: Comparative study of block matching techniques used in video images motions estimation. In: The 5th International Symposium on Image and Signal Processing and Analysis (ISPA'07). (September 2007)
20. Intel: Intel C++ compiler user and reference guides. Intel Document number: 304968-022US (2008)