Accelerating a PARSEC Benchmark Using Portable Subword SIMD

Saugata Ghose, Shreesha Srinath, Jonathan Tse
{*sg532, ss2783, jdt76*}*@cornell.edu*

***Abstract*—We present a case study of the GNU Compiler Collection (GCC) Vector Extensions in GCC 4.7. In particular, we examine the relative performance of explicit vector code using the GCC Vector Extensions to that of automatically vectorized code from the Intel C++ Compiler (ICC). Our analysis focuses on the interactions between data-level and thread-level parallelism in the *streamcluster* benchmark from the PARSEC benchmark suite, in particular examining tradeoffs between portability and performance across different vectorization techniques.**

I. Introduction

An important facet of the computer architecture field is the availability of benchmark suites. In order to evaluate the performance of widely-differing microarchitectural designs and hardware implementations, we use common sets of applications as a basis of comparison. Typically, these applications are representative of the current and/or future application set that computer users will be running.

One such application suite is PARSEC [5], born out of the Intel Platform 2015 initiative [6]. The goal of Platform 2015 was to develop a diverse range of programs that are representative of the most important emerging applications within the next several years. As part of their initiative, Intel and various academics formed a partnership to develop and release these programs as the PARSEC benchmark suite. Currently, PARSEC supports x86-64 and SPARC architectures, and implements parallel support through a combination of OpenMP, pthreads, and the Intel Thread Building Blocks libraries.

Recently, a resurgent direction of research in the computer architecture community is data-parallel architectures. Not only are mainstream workloads focusing more on multimedia-driven processing, but several applications do not scale well in the regime of thread-level parallelism (TLP). To exploit improved performance for both of these families of workloads, data-level parallelism (DLP) can be extracted, despite the limited effectiveness of TLP. Currently, there is limited benchmark support for these DLP-driven architectures, as they tend to require highly-architecture-specific instruction sets.

Results from studies of other applications suites indicate that modern compilers fail to optimize many code patterns and that compiler vectorization performance is quite uneven across compilers [10]. These results illustrate the lack of universal criteria to determine performance critical patterns to vectorize, i.e. explicit vector optimizations should lead to better performance versus auto-vectorized code.

This paper was submitted as part of a final project for CS 5220, *Applications of Parallel Computers*, in Fall 2011 at Cornell University.

We explore the performance of subword SIMD units using the *streamcluster* benchmark from the PARSEC suite. In particular, we explore a hybrid programming model, which combines multithreaded execution with subword SIMD parallelization. While other PARSEC benchmarks use Intel SSE units in serial execution, *streamcluster* does not include any DLP optimizations, making it an appropriate candidate for our analysis of interactions between DLP and TLP. As TLP and DLP are orthogonal, there is a large amount of performance improvement that can potentially be exploited from combining the two.

We use GCC Vector Extensions [8] to implement our subword SIMD code. The GCC Vector Extensions are a portable implementation of vector operations for the C and C++ languages, which are currently under active development. We choose this over a specific subword SIMD architecture to improve the portability of our code, as one of our research objectives is to overcome the specificity of applications to a single instruction set architecture.

One of our major contributions is the exploration and documentation of the current behavior of the GCC Vector Extensions, as information on the extensions is currently difficult to find. We then use this knowledge to add vectorization to *streamcluster*. Finally, as a point of comparison, we examine the effectiveness of the auto-vectorization engine in ICC to our Vector Extension-based efforts.

II. Related Work

The PARSEC benchmark suite has itself been the subject of several studies. In general, across a range of hardware systems, the PARSEC benchmarks are computation-bound, although some are sensitive to memory bandwidth and latency effects [4]. This suggests that spending area and power resources on vector support in hardware is appropriate, either by building vector support at the CPU level, or alternatively by porting the benchmarks to other data-parallel architectures, such as GPUs. The *streamcluster*, *swapoption*, *blackscholes* and *fluidanimate* applications from the PARSEC suite have already implemented using the NVIDIA CUDA framework [12].

In particular, the *streamcluster* benchmark has been evaluated using chip multiprocessor (CMP) systems. Evaluations run on an AMD Opteron-based 8-core multiprocessor system using GCC 4.2.1 and the PIN tool suggest that the maximum speedup for the *streamcluster* application is limited by the serial section of code [5]. Furthermore, in a 32-core x86 system based on the Virtutech Simics simulator, communication made up only 15% of the total execution time of *streamcluster* [3].

TABLE I
INTEL XEON E5504 PROCESSOR SPECIFICATIONS [9]

Frequency	2.0 GHz
Number of cores	4
Cache line size	64 B
IL1/DL1 size	32 KB / 32 KB
IL1/DL1 associativity	4-way / 8-way
L2 size	256 KB, private
L2 associativity	8-way
L3 size	4 MB, shared
L3 associativity	16-way
Vector Extension Set	SSE 4.2
Vector Registers	16
Vector Register Width	128 b

Maleki et al. explore the effects of parallelizing various architecture benchmarks using a number of auto-vectorizing compilers [10]. They propose a number of data reorganizations that are useful for improving the performance of the resultant vectorized code.

III. EXPERIMENTAL SETUP

A. Target Architecture

The tunings described by this paper are targeted for an Intel Xeon E5504 quad-core processor, which is part of the Core i7 family. Our code is restricted to using two of these processors, which are connected together using the Intel QuickPath Interface (QPI). Table I shows the relevant parameters of the processor. The Intel Xeon E5504 use the x86-64 instruction set architecture with SSE 4.2 extensions.

B. Compilers and Software

Several compilers were used for our experiments. Two versions of GCC were used for compilation. The nightly snapshot of GCC 4.7 from November 26, 2011, was used for the majority of our experiments (unless otherwise noted). GCC 4.7 was used as it implements all of the logic and shift operations for the GCC Vector Extensions. We also experimented with GCC 4.6.2, which is the latest stable release of GCC at the time of writing. However, while GCC 4.6 supports targeting the Core i7 architecture, it does not provide full Vector Extension support. The optimization flags used when running GCC are `-O3`, `-funroll-loops`, `-fprefetch-loop-arrays`, and `-march=corei7`.

For experiments which used ICC, we obtained version 12.1. For auto-vectorization, the `-vec-report3` flag was used, which also provided annotations for the vectorization analysis by the compiler. Automatic vectorization was disabled using the `-no-vec` flag. Aside from this, the optimizations used on the benchmarks were `-O3`, `-funroll-loops`, `opt-prefetch`, and `-march=corei7`.

For application profiling, the HPCToolkit was used (version 5.1.0) [1]. The pthread libraries were used for parallelization. All applications were compiled on an Intel Xeon E5405 quad-core processor—part of the Core 2 family.

C. Benchmark Suite

Version 2.1 of the PARSEC benchmarks was used [5]. For our experiments, we look at two sets of configurations: pthread parallelism and serial compilation. Three input sets are studied: *simsmall*, *simlarge*, and *native*. Unless otherwise noted, all results reported are for the *native* inputs, which are the largest of our data sets. While some of the benchmarks include code to exploit the Intel Thread Building Block libraries, we do not utilize them in our experiments.

IV. CURRENT STATE OF VECTORIZATION

A prerequisite for vectorizing an application is the availability of fine-grained parallelism in the data stream. Regularity in both control flow and memory accesses, when present, is a key enabler of vectorization. Vectorizing code, i.e. the employment of subword SIMD methodologies, can be done at several different granularities: *Assembly Language*, *Intrinsics*, *Portable Types and Classes*, and *Auto-Vectorization*. The tradeoff here is the granularity or programmer control over vectorization versus the engineering effort required of the programmer. The following sections present a short discussion of each methodology in order of increasing programmer effort.

A. Assembly Language

Inline assembly code provides direct access to all the instructions and registers available on the target platform, and is limited only by the operations provided by the instruction set architecture extension. Typical use cases include writing assembly-level code for critical code kernels in an application. The effort required of the programmer, as well as the fact that assembly-level programs are not cross-architecture portable, make this methodology unattractive for all but the most performance-critical applications.

B. Intrinsics

Intrinsics are similar to assembly instructions in that they provide access to all the instructions, but there is no need to explicitly allocate registers and handle addressing modes. As a result, this approach integrates well with higher-level languages such as C and C++. While intrinsics are technically portable across processors which support the same ISA, some compilers implement proprietary intrinsic calls, which can place further restrictions on portability in some situations. Furthermore, because intrinsics are ISA-specific, the effort and knowledge-base required of programmers still makes this methodology relatively unattractive.

C. Portable Types and Classes

Some compilers ship with portable vector data types and C++ classes that directly use SIMD instructions. For example, the Intel C++ Compiler provides the vector classes for integer (`I32vec4`), single-precision floating point (`F32vec4`) and double-precision floating point (`F64vec4`) datatypes. These class types provide a high-level interface to SIMD operations. Other third-party SIMD language extensions exist, although their portability is a concern.

In particular, the GCC Vector Extensions—described further in Section V—are more desirable as GCC is supported on many platforms and is freely available. This approach is very attractive as it is portable across all GCC-supported platforms.

However, not all combinations of operations and datatypes are supported, which limits the ability to efficiently implement data-parallelism.

D. Auto-Vectorizing Compilers

All of the previous methods require the programmer or code generator to explicitly expose parallelism, which requires explicit identification and modification of vectorizable code sections in an application. Fortunately, most modern compilers, such as GCC, ICC, and XLC, have the ability to analyze loops in an application and automatically find opportunities to use the SIMD instructions.

Employing compiler-based auto-vectorization techniques is most attractive from the standpoint of minimizing the workload of the programmer. Future generations of compilers can be extended to support new SIMD instructions and different ISAs, so supporting new architectures is as simple as recompiling code.

However, at this time, the auto-vectorization techniques employed by compilers cannot match the performance of code that is hand-tuned by the previously-discussed methodologies. ICC currently represents the best-of-class in auto-vectorization for Intel processors, so we use it as a point of comparison when evaluating the auto-vectorization methodology.

V. GCC Vector Extensions

The C language was originally developed to program SISD (Single Instruction Single Data) architectures. These uniprocessor systems apply scalar operations to scalar data elements and exploit no parallelism in the data elements. In comparison, the subword SIMD vector units on modern platforms execute a single instruction across multiple data elements, which represents a more DLP-centric view of computation.

Various instruction set architectures have been extended to include such streaming operations. The x86 instruction set has the MMX, 3DNow!, SSE extensions for Intel and AMD processors. Altivec is a SIMD instruction set designed by Apple, IBM and Freescale Semiconductor. The MIPS instruction set provides the MDMX, MIPS-3D and MVI extensions. Other examples include the ARM NEON extensions, MAX extensions for PA-RISC, and VIS extensions for the SPARC architectures.

Using vector intrinsics as described in Section IV-B provides a one-to-one mapping with the vector instructions, but restricts the code to one architecture. In part due to its original design parameters, the C language does not natively support vector code without the architecture binding. To address this concern, the GNU Compiler Collection provides generic vector extensions to the C and C++ languages.

A. Portability

The GCC Vector Extensions provided for the C and C++ languages enable the cross-architecture portability for vector code. The vector types for the extensions are defined with a `typedef` with an additional inclusion of an attribute indcating the number of elements in the vector. Code Block 1 declares an integer vector type which is 16 bytes long and contains four integer elements, each of which is four bytes long.

Code Block 1. Integer Vector Type Declaration

```
typedef int v4si
   __attribute__ ((vector_size (16));
```

The allowable base elements for the vector types declared in this manner include the basic C language integer types in both `signed` and `unsigned` formats: `char`, `short`, `int`, `long`, and `long long`. Floating point is also supported for both `float` and `double` types. The only constraint is that the `vector_size` attribute must be a power of two. To ensure portability, if the targeted architecture does not support the specified combination, GCC will synthesize a set of instructions from the more restrictive set to emulate the desired behavior. For example, if a variable of type `v4si` is declared and the target architecture does not allow SIMD execution for this type then GCC will produce code that uses four scalar integers instead.

Often, there is a need to access the individual vector elements. This can be handled by pointer casting, or through the use of `unions`, as shown in Code Block 2.

Code Block 2. Union Type for Signed Integer Vectors

```
typedef union {
  v4si v;
  int i[4];
} VecSignedInt;
```

The `union VecSignedInt` declares the union of a vector of type `v4si` and an array of integer with four elements. The vector type or the individual base element can then be accessed with the dot or arrow operators, similarly to `struct` instances.

Due to the relative readability of the `union` methodology, we employ this throughout our study. GCC 4.7 purports to allow individual element access without resorting to the use of `union` or pointer casting, but we chose not to employ the GCC 4.7 methodology as it is still in development at this time.

While the GCC Vector Extensions provide portability across architectures, they are currently not portable across compilers such as ICC and XLC, both of which may give better architecture-specific performance. A generic vector extension to the C/C++ languages would provide for seamless cross-platform and cross-compiler portability.

The following subsections enumerate the supported C/C++ operations for various vector types and list the constraints under which they are allowed.

B. Arithmetic and Shift Operations

All the basic arithmetic operators (`+, -, *, /, %`) are supported for both integer and vector data types. They are also supported for operations between integer vector and integer scalar data types, whereas they are restricted to only operations between vector data types in the case of floating point. The result of a modulo (`%`) operation must be captured by an integer type of appropriate width and length for floating point vector-vector operations. Furthermore, vector-vector operations must be on vectors of the same type and length.

The shift operations (<<, >>) are only supported for the integer vector datatypes, and they also accept mixed vector- and scalar-operands.

An example that adds two integer vectors (a, b) of the v4si vector type is shown in Code Block 3, along with the GCC-generated assembly instructions for the Intel Core i7 architecture.

Code Block 3. Integer Vector-Vector Add Operation

```
; v4si a, b, c;
;
; c = a + b;

paddd %xmm1,%xmm0
```

GCC 4.7 supports operations between a scalar variable or constant and a vector for integer types only. The example Code Block 4 shows the addition of a, a scalar integer, to the integer vector b. The corresponding assembly instructions show the dynamic creation of an integer vector with all elements initialized to value of a, which is then added to the vector b. As discussed before, combinations of vector and scalar operands are not allowed for floating point vector types.

Code Block 4. Integer Scalar-Vector Add Operation

```
; v4si c, b;
; int a;
;
; c = a + b;

movd %edi,%xmm2
pshufd $0x0,%xmm2,%xmm1
paddd %xmm0,%xmm1
movdqa %xmm1,%xmm0
```

C. Bitwise Operations

The bitwise operations (&, ^, |, ~) are supported for integer vector-vector operations for vectors of the same size. The bitwise vector-vector operations are not supported for the floating point vector types, but casting floating point vectors to integer vectors is a suitable workaround. Code Block 5 shows the C syntax for the bitwise & operation on integer vectors a and b.

Code Block 5. Integer Vector-Vector AND Operation

```
; v4si a, b, c;
;
; c = a & b;

pand %xmm1,%xmm0
```

The C syntax for implementing the floating point bitwise & operation is shown in Code Block 6. The vectors a and b are floating point vector types and the result vector is an integer vector.

Code Block 6. Float Vector-Vector AND Operation

```
; v4sf a, b;
; v4si c;
;
; c = (v4si)(a) & (v4si)(b);

pand %xmm1,%xmm0
```

As with the arithmetic operations, GCC 4.7 only supports operations between integer scalar and vector operands. Code Block 7 shows the bitwise & operation between a scalar integer, a, and an integer vector, b.

Code Block 7. Integer Scalar-Vector AND Operation

```
; v4si c, b;
; int a;
;
; c = a & b;

movd %edi,%xmm2
pshufd $0x0,%xmm2,%xmm1
pand %xmm1,%xmm0
```

D. Relational Operations

The relational operations (==, !=, <, <=, >, >=) are supported for all expressions where the operands are vector types of the same dimension. The operations are also supported between integer vectors and integer scalars. Regardless of the operand types, the result of the comparison is a vector of the same length as the operands of the signed integer vector type. The element-wise comparison results are 0 for false (all bits 0) or -1 for true (all bits 1).

The C syntax and the assembly for the greater than equal to comparison of integer vectors a and b ($a \geq b$) are as shown in Code Block 8. GCC 4.7 currently does not support comparison operations between scalar and vector operations for floating point vectors.

Code Block 8. Integer Vector-Vector Compare Operation

```
; v4si a, b, c;
;
; c = a >= b;

pcmpgtd %xmm0,%xmm1
movdqa %xmm1,%xmm0
pcmpeqd %xmm1,%xmm1
pandn %xmm1,%xmm0
```

E. Shuffle Operations

Shuffle operations are used to reorganize the layout of data in preparation for SIMD execution. These operations are often used to convert arrays of structures to structures of arrays, which are more friendly for SIMD memory operation. Operating on the individual elements, while possible, is not efficient.

Vector shuffle operations are supported via builtin functions. There are two variants of shuffle operations, which can be applied to any vector type. The first variant shuffles the elements of a vector given a mapping, specified as an integer vector of the same length. The C syntax in Code Block 9 shows the shuffle operation to reverse the order of elements in the integer vector a.

Code Block 9. Shuffle Single Vector

```
v4si a, c;
v4si mask = {3, 2, 1, 0}

c = __builtin_shuffle(a, mask);
```

The shuffle operation can also be applied to two vectors, also using a mapping. The elements of the first vector begin at 0 and the second vector is considered to begin at N, where

N is the width of the vectors. The mapping will undergo a modulo by $2N$ before being used.

The example in Code Block 10 creates a new integer vector, `c`, which has all the even elements selected from the input integer vectors `a` and `b` using the appropriate mapping.

Code Block 10. Shuffle Two Vectors

```
v4si a, b, c;
v4si mask = {1, 3, 5, 7}

c = __builtin_shuffle(a, b, mask);
```

F. Current Limitations

Currently, GCC 4.7 supports a limited subset of the Vector Extensions for the C++ language. The vector-vector operands are supported for all arithmetic and bitwise logical operations for integer and floating point types whereas the shift and relational operations are not supported for vector operands. Furthermore, in C++, none of the scalar-vector operations are supported, even for integer data types. There are of course workarounds, but all workarounds incur some additional overheads. The lack of support for these operations restricts the portability of code across the C and C++ code. This shortcoming of GCC 4.7 is likely due to the fact that is in active development.

Embracing the generic vector extensions limits the programmer to the operations and `builtin` functions provided by GCC. Other vector extensions provide useful operations currently unsupported by GCC. For example, the SSE intrinsics provide a horizontal summation operation for the elements of a vector. However, the *swizzle* functionality needed from SIMD programming is emulated by the shuffle `builtin` functionality of the GCC Vector Extensions. Interestingly, the semantics of the `__builtin_shuffle` operation are similar that of the OpenCL `shuffle` and `shuffle2` functions.

Further extensions of the available `builtin` functions and various types will increase the utility of GCC as a productive, portable SIMD toolbox to the programmer. We hope to see such improvements in future GCC releases.

VI. Case Study: Streamcluster

The *streamcluster* PARSEC benchmark represents an important kernel used in data mining algorithms, which solves the online clustering problem. It precludes algorithms that require random access or a large memory footprint. The original problem of clustering is NP-hard, but the implementation in *streamcluster* uses a heuristic solution [11].

The *streamcluster* benchmark is particularly interesting to our study because it has moderate thread-level parallelism, due to significant amounts of synchronization. However, the kernel has low levels of data sharing between threads, and most sharing between threads is read-only. Each of the input streams for the kernel undergoes iterative computation, where the same operation is applied to the varying set of input points present in each stream. This is a good fit for the data-parallel subword SIMD model since it is computationally intense and does not have a dependence between points within an input stream.

A. Application Overview

At its heart, the *streamcluster* kernel follow a data stream model where each of the inputs are processed in the order they arrive. The inputs to the kernel are generated within the kernel for a selected input stream chunk size. A chunk represents a collection of input points within a current input stream along with the attributes associated with each point. Each point is represented by a structure which includes the weight, assigned cluster number, and coordinates for a specified number of input dimensions.

The input stream is first transformed to an intermediate input frequency weighted representation based on the number of discrete values present in the current chunk. The algorithm solves the clustering problem for the current input chunk. It then compares all the clusters obtained thus far and solves a variant of the clustering problem called the facility location, which reorganizes the cluster by considering the minimum and maximum number of cluster centers allowed [11].

B. Profiling Results

We used HPCToolkit to profile the *streamcluster* kernel. We ran the application in several configurations, varying the input dataset and the number of threads independently. The HPCToolkit reported that 95% of the execution time is spent in the `distance` function, which computes the Euclidean distance between two points. Although the function is called multiple times, we believe that HPC overestimates the time spent in the `distance` function.

VII. Distance Microkernel

We choose to examine the `distance` function for a number of reasons. Not only does it seem to take up a large portion of the execution time, but its data access pattern also allows us to investigate how memory layout affects the performance of vectorization. From observation, we also determined that several opportunities for vectorization within the benchmark have similarities to the structure of this function. As such, it is an excellent candidate for use in microkernel experiments.

We will first explore several different naive approaches to vectorizing the microkernel. We then apply several optimizations to each of the approaches to address shortcomings observed in the naive kernel assembly code. We compare both our original and optimized implementations against the scalar version of the microkernel. We present both results in order to demonstrate the potential payoff of and the need to highly optimize vector code.

In order to attribute any overhead directly to the vectorization of these kernels, we assume that all inputs are evenly divisible by our vector width of four. We also assume that vectorization is only being performed in certain sections of the code, and that therefore all datatypes that exist outside of the actual microkernel function are scalar. This requires all vector assembly and casting to be done inside the microkernel. All dynamic memory allocation that is not data-dependent is performed outside of the microkernel, including any vector types.

A. Potential Vectorization Methods

Code Block 11 shows the original `distance` function, modified to loop over all of the particles within the function. (Inside the benchmark, the `distance` function is always called from within a loop which iterates over several points.) Each point consists of an array of `float` objects. For each point `p1` paired with the reference point `p2`, the function iterates through all of its dimensions, one at a time, calculating the square of the difference and accumulating the value.

Code Block 11. The `distance` microkernel.

```
1  void dist(float* distance, Points src) {
2    unsigned int i, j;
3
4    for(j = 0; j < src.num; ++j) {
5      for(i = 0; i < src.dim ; ++i) {
6        distance[j] +=
7            (src.p[j].coord[i] - src.p[0].coord[i]) *
8            (src.p[j].coord[i] - src.p[0].coord[i]);
9      }
10   }
11 }
```

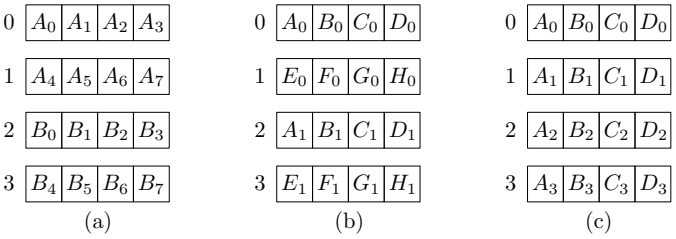

Fig. 1. First four loop iterations for various `distance` microkernel vectorization approaches. For this diagram, we assume that a coordinate N_i represents dimension i of point N. The above example shows the vectorization behavior for eight points, A through H, and eight dimensions for three loop iterations. The above iterations are for: (a) *DistDimensions*, (b) *DistPoints*, and (c) *Dist4Points*.

Our first approach, *DistDimension*, simply vectorizes the original function to operate on four dimensions each interval. We need to accumulate these values, so at the end of each interval, we perform a reduction using addition and then add this to a scalar accumulation variable. Since we do not have to alter the data structure, we simply use a vector pointer to traverse over the array as if it is of type `v4sf`.

Our second approach, *DistPoints*, alters the data arrangement significantly. As seen in Figure 1, we reorganize the elements so that we maintain an array for each dimension. Each array contains the coordinates for all points *for the given dimension*. Arranging the data in such a manner provides us a significant advantage over *DistDimension*: we no longer have to perform a reduction. Instead, we can perform addition in parallel, though this comes at the price of having to pack four copies of each dimension for point `p2` into a single vector. As this duplication cannot be simply read out of the array, we must pay a penalty for each dimension that we examine.

One major issue with this algorithm is that we lose cache locality on the accumulation vector, as we must revisit the result vector of every point each dimension. To alleviate this, we propose a third method, *Dist4Points*, which maintains the data organization by dimension, but only processes the results for four points at a time. While this maintains the parallelism of the addition, we can exploit locality for the accumulation vector, by never having to revisit it. However, this comes at the cost of locality for the dimension vectors, as we only read 16 of the 64 bytes pulled into the cache line each iteration. We then jump to another line, again to read only 64 bytes. In reality, this is worsened by the fact that the prefetcher likely pulls in subsequent lines of the array, which we also ignore. However, we would like to compare this performance to see which locality is more pertinent.

TABLE II
INPUTS USED FOR MICROKERNEL EVALUATION

Input Set	Description	Points	Dimensions
D	Large number of dimensions, small number of points	16	1024
P	Large number of points, small number of dimensions	1024	16
PDS	Equal number of points and dimensions, small size	128	128
PDL	Equal number of points and dimensions large size	1024	1024
NTV	Number of points and dimensions found in PARSEC *native* input set	16	1024

B. Method Optimizations

Our initial implementations of the code naively convert the `distance` microkernel. Upon studying the underlying assembly of the code, we find several improvements that we can make to each version of the code, in order to improve performance.

For *DistDimension*, we eliminate the need to perform the reduction at every iteration. Instead of maintaining a single accumulation variable, we instead keep an entire accumulation vector, where each lane accumulates its own sum. We only perform a reduction of this accumulation vector at the very end of the inner loop, when we need to obtain the distance.

We perform a couple of smaller optimizations within the *DistPoints* code. Instead of packing the dimension vector for point `p2` one element at a time, we instead use an array initialization to assign the whole vector at once. We also use pointer arithmetic to avoid having to perform complex arithmetic operations within an array index. We find that both of these optimizations do help in decreasing the overall size of the assembly code.

Dist4Points sees a significant change. Instead of packing each dimension vector for point `p2` every loop iteration, we now construct the whole dimension vector, containing four duplicates of each dimension coordinate, so we do not have to incur as expensive an overhead. Again, pointer arithmetic is used to reduce the need to perform a large number of memory address calculations.

C. Performance

We use a series of inputs, shown in Table II, to determine the behavior of these microkernels under different input conditions. Figure 2 shows the performance of the unoptimized microkernels. The observed speedups are, for the most part, unremarkable. While the PDS and PDL inputs show a speedup of approximately 2.15 for the *DistPoints* mechanism, it is not the clear winner, as the effects of poor caching in that scheme

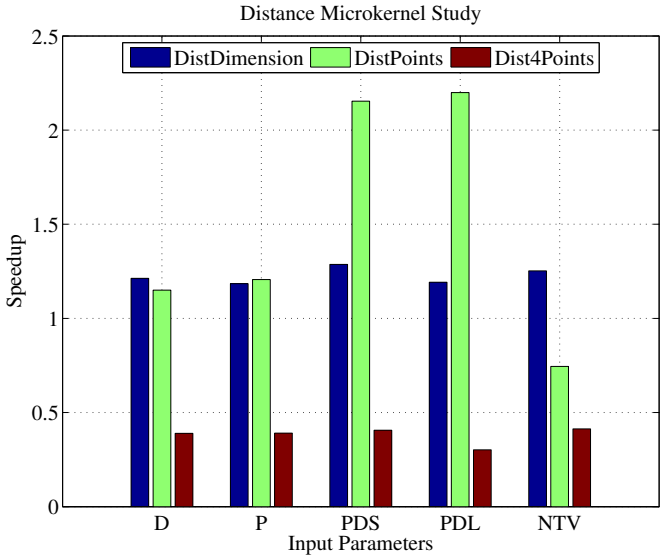

Fig. 2. Speedup of `distance` microkernel vs. scalar function with naive vectorized implementations.

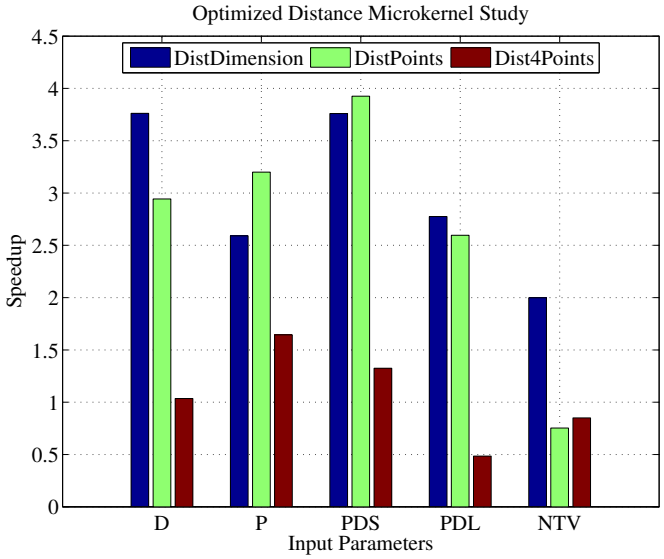

Fig. 3. Speedup of `distance` microkernel vs. scalar function after code optimizations are performed.

are magnified in the NTV input set, and we see performance significantly worse than the serial implementation. In contrast, the *DistDimension* shows consistent speedups across all of the input sets, but they are decidedly lackluster, as the constant need for reductions reduces most of the vectorization benefits. The *Dist4Points* shows abysmal performance, never improving over a slowdown of 2 versus the original code. Unfortunately, the poor utilization of the cached dimension vectors is taking a large toll.

Upon optimization, we see much more encouraging numbers for performance. Indeed, for some inputs, we approach a speedup for 4, which is the theoretical upper performance bound for our 4-word subword SIMD units. Again, we see that *DistPoints* does poorly in the NTV input set, due to the cache behavior. Our optimizations are unable to improve that performance significantly. We find that *DistDimension* performs remarkably well across all input sizes. However, it is also apparent that as the number of points increases, we experience diminishing returns for the vectorized speedup.

While we finally see some speedups for *Dist4Points*, the optimized version still leaves much to be desired. We also experimented with increasing the vector widths of the struct, and letting GCC unroll the vector inside the innermost loop to exploit the full cache line at each iteration. What we find is that this does not help us significantly. The cache utilization improvements come at the cost of increased register pressure, and as a result, to use four sets of vectors, we must constantly move values in and out of the XMM registers within a single iteration, adding significant overhead.

Due to its excellent performance, as well as the lack of major structural reorganization within the code base, we choose to implement the *DistDimension* microkernel version within our final benchmark code.

VIII. GetCost Microkernel

We select a second microkernel benchmark to investigate a different aspect of the vectorization problem. While the `distance` microkernel investigates the memory arrangement properties, it implements a simple nested `for` loop to perform this. We therefore look to the `pspeedy` function to find a microkernel with a conditional assignment. A vectorized conditional loop requires a non-obvious solution. We present three approaches to implementing such a loop, and evaluate their associated overheads.

A. Potential Vectorization Methods

The original microkernel is shown in Code Block 12. We make a couple of simplifications to the original code, in order to hone in on only the effects of vectorization on microkernel performance. First, we replace the call to the `distance` function with a random number generator, to reduce cache pressure and to eliminate the function call overhead. Second, we also assume that the number of dimensions is fixed at 128. This is because, aside from the initialization of the `struct` array, we never need to access the `coord` array—which contains the coordinates for each dimension—within our microkernel.

Code Block 12. The `getCost` microkernel.

```
1  void getCost(Points *points) {
2    for(int i = 1; i < points->num; ++i) {
3      for(int k = 0; k < points->num; ++k) {
4        float distance = ((float)(lrand48()) /
5              (float)(INT_MAX));
6
7        if(distance*points->p[k].weight <
8              points->p[k].cost) {
9          points->p[k].cost = distance *
10           points->p[k].weight;
11         points->p[k].assign = i;
12       }
13     }
14   }
15 }
```

To vectorize the code efficiently, we cannot access various variables across non-unit strides, as we will experience significant performance degradation from having to pack vectors each time the microkernel is invoked. Therefore, we assume that the `cost`, `weight`, and `assign` variables have been taken out of the `Point` structure, and have been consolidated as contiguous arrays in the `Points` structure. In fact, we find that we will need to make such a conversion inside *streamcluster* for most of our vectorization changes.

The first optimization, *Mask*, performs vectorization by using a vectorized comparison result as a mask. We use the mask and its inverse to clear the unwanted data from each of the two vectors to be combined. We then use a logical OR to combine them into a single vector.

For *Shuffle*, our second approach, we take the comparison vector, and instead of using it as a mask, calculate an offset. For clarity, recall that a vectorized comparison will place a value of 0 when a comparison is false for the particular vector element, and a -1 when the comparison is true. If we treat this comparison vector as a packed signed integer, we can perform some calculations on it to translate it into a shuffle mask. We simply multiply it by the vector width and then add the offsets for each position inside the vector. Note that we also have to add the vector width back in, to allow only non-negative mask indices. We then feed this into the shuffle function, which assembles the result vector from the two input vectors.

Bitwise, our third approach, employs a "bit-twiddling" trick from the graphics community [2]. Again using the comparison vector, we can perform an XOR-AND-XOR operation to accomplish the same masking that *Mask* does logically, but in fewer instructions.

B. Performance

One unknown in our implementations was the cost of the instruction overhead needed to move values to and from the XMM registers. Unfortunately, as we see in Figure 4, these overheads outweigh any performance improvements. We see that *Mask* and *Bitwise* perform similarly, with *Bitwise* always having a slight edge due to its smaller instruction count. *Shuffle* performs significantly worse, due to the fact that the GCC function `__builtin_shuffle` expands into a seven-instruction sequence. As a result, its overhead is high to be a viable option.

While the results here are somewhat disappointing, they motivate the need to investigate a larger vector width. With larger numbers of packed integers, we will be able to better overcome the register copy overhead, and will be able to add only a few more instructions to take the place of double the code. Our vectorized code was replacing a scalar loop that, for four entries, required sixteen instructions. While we see Intel finally adopting this trend with their move to the AVX instruction set—which has 256-bit registers, it is only quite recent, and our current hardware unfortunately does not support AVX instructions. As a result, we were unable to confirm this hypothesis experimentally.

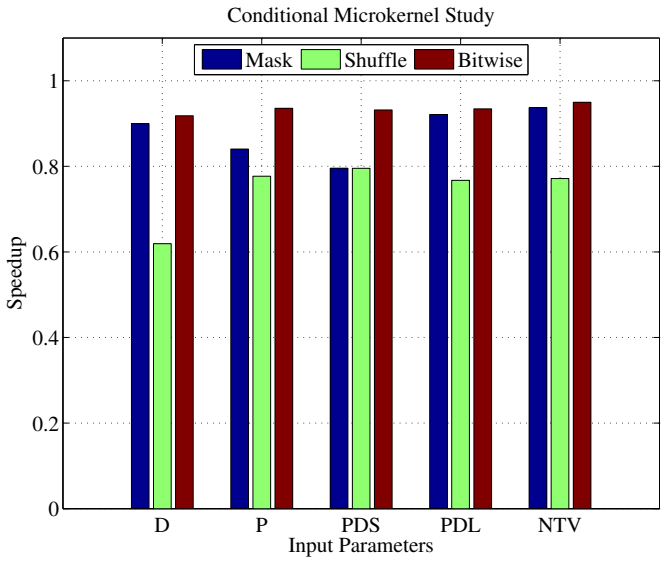

Fig. 4. Speedup of `getCost` microkernel vs. scalar function.

IX. Evaluation

We apply what we learned from the microkernels to the *streamcluster* benchmark, vectorizing the application where see find potential for improvements. As mentioned in Section VIII-A, we had to rearrange the data layout to better provide vectorization opportunities at a low overhead. We study four versions of the benchmark:

- *streamcluster*—the original benchmark,
- *scstruct*—the benchmark with the altered data layout, but with no vectorization within the program,
- *scdist*—the benchmark with no changes to its data structures and the vectorized `distance` microkernel added in, and
- *scvector*—the benchmark with the altered data layout and all vectorization added in.

The *scstruct* version allows us to determine what performance effects are observed from the modified data layout. We extracted the `distance` microkernel into its own version to see if we could validate the observation of the profiler, as well as to observe the effects of our non-microkernel vectorizations independently.

We found two caveats during our vectorization implementation. In the original code, the `assign` value was of type `long`, which in a 64-bit system is 8 bytes long. Since the SSE instructions do not support packed operations on `long`s, we converted the `assign` value to be an `unsigned int` instead. We find that this has no bearing on the correctness of the code, and that it works for all of our input sets with significant room for growth.

The second, and more troublesome, issue was that in our nightly build of GCC 4.7, we found that while `gcc` supports all of the tested Vector Extension instructions, `g++` unfortunately does not have support yet for all of them (see Section V-F). As a result, while we could vectorize non-conditional loops, we were unable to do conditional loops like our `getCost` microkernel, since the bitwise operators were unavailable.

TABLE III
SPEEDUP FOR SERIAL VERSIONS OF VECTORIZED PROGRAMS OVER BASELINE SERIAL GCC, NATIVE INPUT SET

Version	Speedup
scstruct	0.83
scdist	1.55
scvector	1.49
icc-streamcluster	1.41
icc-scstruct	1.42

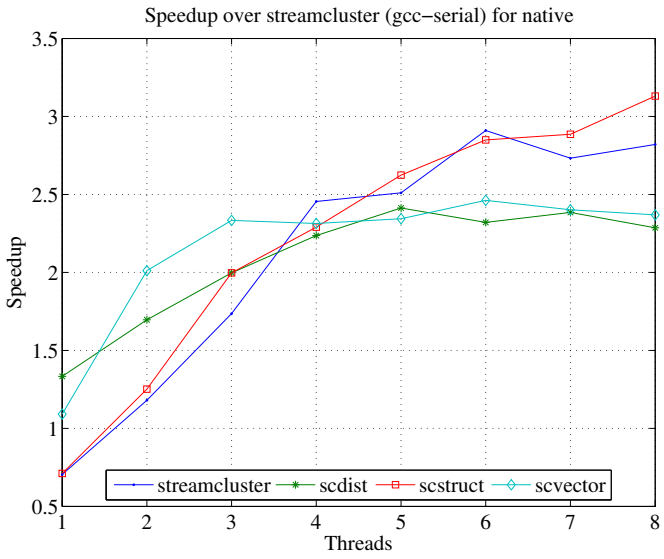

Fig. 5. Speedup over serial *streamcluster* for the *native* data set.

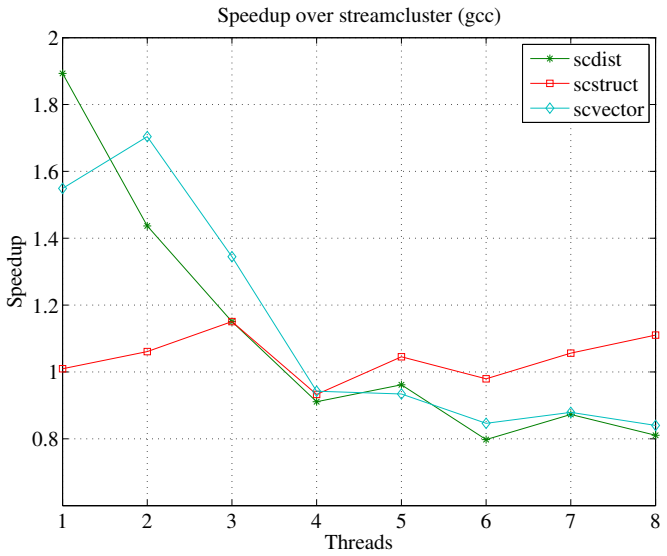

Fig. 6. Speedup from vectorization over the parallel version of *streamcluster*. Note that for each thread count, speedup is shown relative to the unmodified benchmark *run with that same number of threads*.

A. GCC Vector Extension Implementation

During our static analysis, we found eight loops that we were able to vectorize. Table III shows the speedup from vectorization without any parallel overheads. We see that *scdist* performs significantly better than the baseline. Using a rough calculation, if we use the 2x speedup obtained from the microkernel results (see Section VII-C), and assuming that there are no overheads due to cache access patterns, we can use Amdahl's law to derive that the `distance` function takes up 71% of the execution time—a high amount, but not as high as reported. We also observe that, without any vectorization, *scstruct* shows a performance degradation due to the rearrangement of data, suggesting a potential decrease in how the caches are exploiting locality.

Figure 5 shows the speedups observed for the *native* input set, compared to the baseline serial performance. We see that while *scdist* does better for the single thread version, *scvector* outperforms *scdist* when multiple threads are invoked, showing that our other vectorizations do have a positive effect on the performace despite the fact that `distance` takes up the majority of execution time.

What we notice is that for more than four threads, we actually perform worse than the original benchmark. This is interestingly the point at which we also observe sub-linear scaling for the *streamcluster* application. What we surmise is that, due to the nature of the application, we saturate the amount of exploitable parallelism. We can see this more clearly in Figure 6, which shows the per-thread speedup that results strictly from vectorization.

While this explains why our vectorization has no tangible benefit, there must be other underlying causes that lead to a decrease in performance. We suspect that this is due to a change in the parallel interactions of our threads. The benchmark uses a simplistic barrier implementation, which does not reduce coherence traffic significantly when a thread is spinning on the barrier, aside from exponential backoff. What could be happening is that with the threads finishing faster, more threads arrive at the barrier earlier, inadvertently increasing contention. It is well-known that increased thread contention results in longer wait times at the barrier [7]. This additional coherence traffic might be compounded by the fact that inter-processor traffic is sent on the QPI link, which has a longer latency than the L3 cache access time between cores on the same processor. Furthermore, due to the size of the *native* input set, distributing work amongst larger numbers of threads may lead to a greater number of memory bank conflicts, which will slow down memory scheduling (and therefore execution time).

Figures 7 and 8 show the speedups obtained for the smaller data sets *simlarge* and *simsmall*, respectively. What we observe is that as the data size decreases, our improvements over the baseline increase significantly. While the speedups are modest in the case of *simlarge*, *simsmall* shows significant speedups from vectorization. While much of that is due to our improved microkernel performance for smaller input sets, part of that is also due to the fact that at smaller thread counts, vectorization can exploit DLP that a multithreaded approach simply cannot take advantage of, due to the parallelization overhead.

B. Auto-Vectorization

Table III and Figure 9 show how simply using ICC with auto-vectorization performs in comparison with GCC. We see that without auto-vectorization, ICC actually performs worse than GCC. However, with it, ICC shows a notable

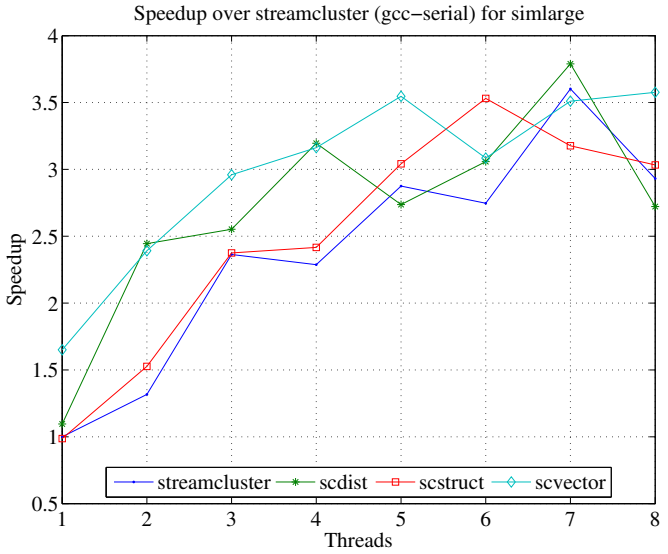

Fig. 7. Speedup over serial *streamcluster* for the *simlarge* data set.

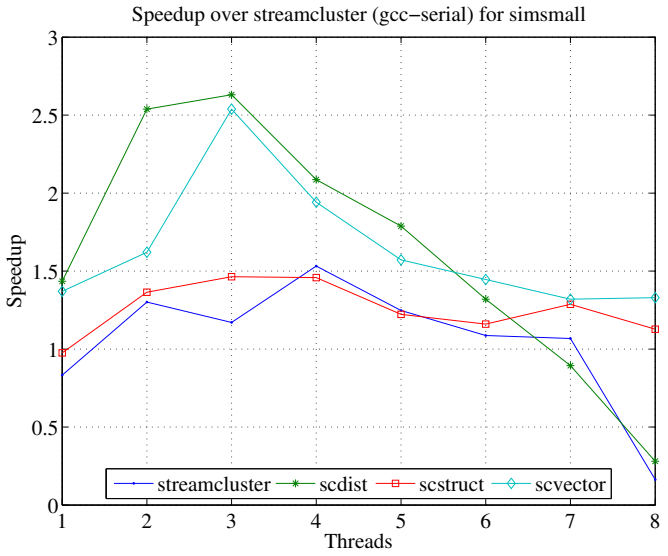

Fig. 8. Speedup over serial *streamcluster* for the *simsmall* data set.

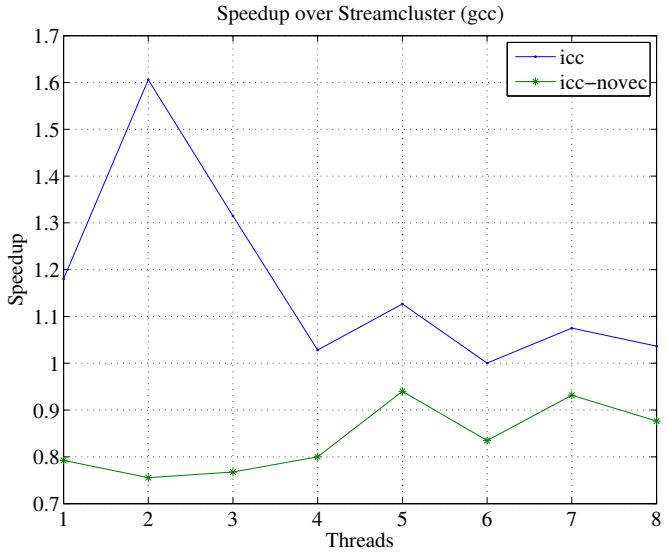

Fig. 9. Speedup of *icc* over *gcc* for the *streamcluster* benchmark.

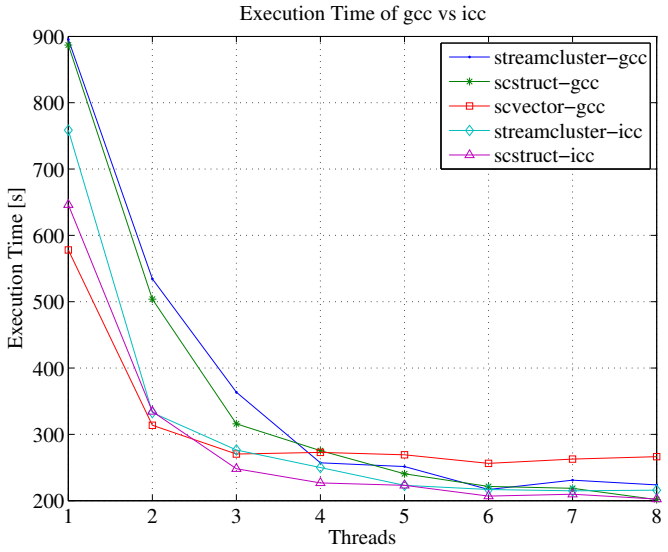

Fig. 10. Execution time comparison across `gcc` and *icc* auto-vectorization schemes.

improvement for smaller numbers of threads. Again, we see that for four or more threads, the benefits of vectorization are negligible.

We compare our manual vectorization efforts against those of ICC in Figure 10. We see that for small thread counts, our portable vectorization efforts perform comparably to the automatic vectorization exploited by ICC. However, unlike our effort, the vectorized ICC versions do not perform any worse than the baseline code for larger numbers of threads. We are currently unsure of what the difference is, as there is little information available about how the ICC vectorizer generates its machine code. However, some of these benefits may be from the reduction of memory bank conflicts by the compiler. As an aside, Figure 11 better illustrates that the ICC auto-vectorization can exploit some additional DLP when we provide it with the data structure rearrangement in *scstruct*. This reinforces the fact that auto-vectorizing compilers can still benefit from some programmer guidance, though we see that the improvement is not very large.

X. Conclusion

The GCC Vector Extensions provide a portable interface for implementing subword SIMD parallelism within C and C++ programs. We implemented two microkernels to examine the performance of these extensions, and used these observations to vectorize the *streamcluster* benchmark from the PARSEC suite. What we find is that for smaller numbers of threads, we outperform the auto-vectorization of ICC, which is best-in-class. However, for high levels of TLP, we perform worse than the un-vectorized code. This is likely due to parallelization overheads and memory conflicts.

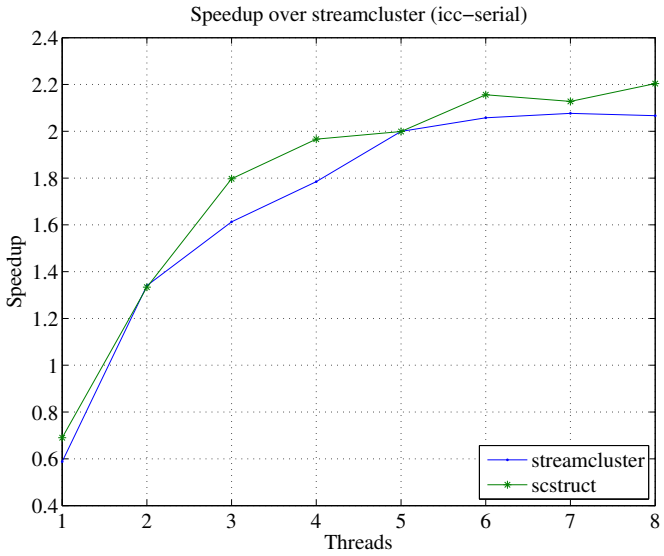

Fig. 11. Speedup over serial *streamcluster*, compiled with *icc*.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. "HPCToolkit: Tools for performance analysis of optimized parallel programs." *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Sean E. Anderson. "Bit Twiddling Hacks." 2005. http://graphics.stanford.edu/ seander/bithacks.html

[3] Nick Barrow-Williams and Christian Fensch and Simon Moore. "A Communication Characterization of SPLASH-2 and PARSEC." In *Proc. of the International Symposium on Workload Characterization*, October 2009.

[4] Major Bhadauria, Vincent M. Weaver and Sally A. McKee. "Understanding PARSEC Performance on Contemporary CMPs." In *Proc. of the International Symposium on Workload Characterization*, October 2009.

[5] Christian Bienia and Kai Li. "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors." In *Proc. of the Fifth Annual Workshop on Modeling, Benchmarking and Simulations*, June 2009.

[6] Shekar Y. Borkar et al. "Platform 2015: Intel Processor and Platform Evolution for the Next Decade." Intel Corp. (*white paper*), March 2005.

[7] Jie Chen and William Watson III. "Software Barrier Performance on Dual Quad-Core Opterons." In *Proc. of the International Conference on Networking, Architecture, and Storage*, 2008.

[8] Free Software Foundation, Inc. "Vector Extensions - Using the GNU Compiler Collection (GCC)." 2011. http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

[9] Intel Corp. "Intel Xeon Processor 5500 Series Specification Update." April 2011.

[10] Saeed Maleki, Yaoqing Gao, Maria J. Garzaran, Tommy Wong and David Padua. "An Evaluation of Vectorizing Compilers." In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, October 2011.

[11] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, R. Motwani "Streaming-data algorithms for high-quality clustering." In *Proceedings of 18th International Conference on Data Engineering*, pages 685-694, 2002.

[12] Matthew D. Sinclair, Henry Duwe, and Karthikeyan Sankaralingam. "Porting CMP Benchmarks to GPUs." University of Wisconsin-Madison Technical Report (TR 1693), June 2011.